IBM WebSphere eXtreme Scale Version 7.0

Programming Guide

March 11, 2011

IBM

This edition applies to version 7, release 0, of WebSphere eXtreme Scale and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About the *Programming Guide*

The WebSphere® eXtreme Scale documentation set includes three volumes that provide the information necessary to use, program for, and administer the WebSphere eXtreme Scale product.

WebSphere eXtreme Scale library

The WebSphere eXtreme Scale library contains the following books:

- The *Administration Guide* contains the information necessary for system administrators, including how to plan application deployments, plan for capacity, install and configure the product, start and stop servers, monitor the environment, and secure the environment.
- The *Programming Guide* contains information for application developers on how to develop applications for WebSphere eXtreme Scale using the included API information.
- The *Product Overview* contains a high-level view of WebSphere eXtreme Scale concepts, including use case scenarios, and tutorials.

To download the books, go to the WebSphere eXtreme Scale library page.

You can also access the same information in this library in the WebSphere eXtreme Scale information center.

Who should use this book

This book is intended primarily for application developers.

How this book is structured

The book contains information about the following major topics:

- **Chapter 1** includes information about getting started withWebSphere eXtreme Scale.
- **Chapter 2** includes information about how to program WebSphere eXtreme Scale.
- **Chapter 3** includes information about accessing data.
- **Chapter 4** includes information about System APIs and plug-ins.
- **Chapter 5** includes information about integrating with the Spring framework.
- **Chapter 6** includes information about the security API.
- **Chapter 7** includes information about the administration API.
- **Chapter 8** includes information about performance considerations.
- **Chapter 9** includes information about troubleshooting.
- **Chapter 10** includes the product glossary.

Getting updates to this book

You can get updates to this book by downloading the most recent version from the WebSphere eXtreme Scale library page.

How to send your comments

Contact the documentation team. Did you find what you needed? Was it accurate and complete? Send your comments about this documentation by e-mail to wasdoc@us.ibm.com.

Chapter 1. Getting started with WebSphere eXtreme Scale

After you install WebSphere eXtreme Scale in a stand-alone environment, use the following steps as a simple introduction to its capability as an in-memory data grid.

The stand-alone installation of WebSphere eXtreme Scale includes a sample that you can use to verify your installation and to see how a simple eXtreme Scale data grid and client can be used. The getting started sample is in the `*installRoot*/ObjectGrid/gettingstarted` directory.

The getting started sample provides a quick introduction to eXtreme Scale functionality and basic operation. The sample consists of shell and batch scripts designed to start a simple grid with very little customization needed. In addition, a client program, including source, is provided to run simple create, read, update, and delete (CRUD) functions to this basic grid.

Scripts and their functions

This sample provides the following four scripts:

The `env.sh|bat` script is called by the other scripts to set needed environment variables. Normally you do not need to change this script.

- UNIX Linux `./env.sh`
- Windows `env.bat`

The `runcat.sh|bat` starts the eXtreme Scale catalog service process on the local system.

- UNIX Linux `./runcat.sh`
- Windows `runcat.bat`

The `runcontainer.sh|bat` script starts a container server process. You can run this script multiple times with unique server names specified to start any number of containers. These instances can work together to host partitioned and redundant information in the data grid.

- UNIX Linux `./runcontainer.sh` *unique_server_name*
- Windows `runcontainer.bat` *unique_server_name*

The `runclient.sh|bat` script runs the simple CRUD client and starts the given operation.

- UNIX Linux `./runclient.sh` *command value1 value2*
- Windows `runclient.sh` *command value1 value2*

For *command*, use one of the following options:
- Specify as `i` to insert *value2* into data grid with key *value1*
- Specify as `u` to update object keyed by *value1* to *value2*
- Specify as `d` to delete object keyed by *value1*
- Specify as `g` to retrieve and display object keyed by *value1*

Note: The *installRoot*/ObjectGrid/ gettingstarted/src/Client.java file is the client program that demonstrates how to connect to a catalog server, obtain an ObjectGrid instance, and use the ObjectMap API.

Basic steps

Use the following steps to start your first data grid and run a client to interact with the data grid.

1. Open a terminal session or command line window.
2. Use the following command to navigate to the `gettingstarted` directory:

   `cd installRoot/ObjectGrid/gettingstarted`

   Substitute *installRoot* with the path to the eXtreme Scale installation root directory or the root file path of the extracted eXtreme Scale trial *installRoot*.
3. Set or export the JAVA_HOME environmental variable to reference a valid JDK or JRE Version 1.5 or later installation directory.

   UNIX Linux export JAVA_HOME=*Java_home_directory*

   Windows set JAVA_HOME=*Java_home_directory*
4. Run the following script to start a catalog service process on localhost:
   - UNIX Linux `./runcat.sh`
   - Windows `runcat.bat`

   The catalog service process runs in the current terminal window.
5. Open another terminal session or command line window, and run the following command to start a container server instance:
   - UNIX Linux `./runcontainer.sh server0`
   - Windows `runcontainer.bat server0`

   The container server runs in the current terminal window. You can repeat step 5 and 6 if you want to start more container server instances to support replication.
6. Open another terminal session or command line window to run client commands.
   - Add data to the data grid:
     – UNIX Linux `./runclient.sh i key1 helloWorld`
     – Windows `runclient.bat i key1 helloWorld`
   - Search and display the value:
     – UNIX Linux `./runclient.sh g key1`
     – Windows `runclient.bat g key1`
   - Update the value:
     – UNIX Linux `./runclient.sh u key1 goodbyeWorld`
     – Windows `runclient.bat u key1 goodbyeWorld`
   - Delete the value:
     – UNIX Linux `./runclient.sh d key1`
     – Windows `runclient.bat d key1`
7. Use <ctrl+c> to stop the catalog service process and container servers in the respective windows.

Defining an ObjectGrid

The sample uses the `objectgrid.xml` and `deployment.xml` files that are in the *installRoot*/ObjectGrid/gettingstarted/xml directory to start a container server. The `objectgrid.xml` file is the ObjectGrid descriptor XML file and the `deployment.xml` file is the ObjectGrid deployment policy descriptor XML file. Both files together define a distributed ObjectGrid topology.

ObjectGrid descriptor XML file

An ObjectGrid descriptor XML file is used to define the structure of the ObjectGrid that is used by the application. It includes a list of BackingMap configurations. These BackingMaps are the actual data storage for cached data. The following example is a sample `objectgrid.xml` file. The first few lines of the file include the required header for each ObjectGrid XML file. This example file defines the Grid ObjectGrid with Map1 and Map2 BackingMaps.

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="Grid">
            <backingMap name="Map1" />
            <backingMap name="Map2" />
        </objectGrid>
    </objectGrids>

</objectGridConfig>
```

Deployment policy descriptor XML file

A deployment policy descriptor XML file is passed to an ObjectGrid container server during start-up. A deployment policy must be used with an ObjectGrid XML file and must be compatible with the ObjectGrid XML that is used with it. For each objectgridDeployment element in the deployment policy, you must have a corresponding ObjectGrid element in your ObjectGrid XML. The backingMap elements that are defined within the objectgridDeployment element must be consistent with the backingMaps found in the ObjectGrid XML. Every backingMap must be referenced within one and only one mapSet.

The deployment policy descriptor XML file is intended to be paired with the corresponding ObjectGrid XML, the `objectgrid.xml` file. In the following example, the first few lines of the `deployment.xml` file include the required header for each deployment policy XML file. The file defines the objectgridDeployment element for the Grid ObjectGrid that is defined in the `objectgrid.xml` file. Both the Map1 and Map2 BackingMaps that are defined within the Grid ObjectGrid are included in the mapSet mapSet that has the numberOfPartitions, minSyncReplicas, and maxSyncReplicas attributes configured.

```
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy
 ../deploymentPolicy.xsd"
 xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">

    <objectgridDeployment objectgridName="Grid">
        <mapSet name="mapSet" numberOfPartitions="13" minSyncReplicas="0" maxSyncReplicas="1" >
            <map ref="Map1"/>
            <map ref="Map2"/>
        </mapSet>
    </objectgridDeployment>

</deploymentPolicy>
```

The numberOfPartitions attribute of the mapSet element specifies the number of partitions for the mapSet. It is an optional attribute and the default is 1. The number should be appropriate for the anticipated capacity of the grid.

The minSyncReplicas attribute of mapSet is to specify the minimum number of synchronous replicas for each partition in the mapSet. It is an optional attribute and the default is 0. Primary and replica are not placed until the domain can support the minimum number of synchronous replicas. To support the minSyncReplicas value, you need one more container than the value of minSyncReplicas. If the number of synchronous replicas falls below the value of minSyncReplicas, write transactions are no longer allowed for that partition.

The maxSyncReplicas attribute of mapSet is to specify the maximum number of synchronous replicas for each partition in the mapSet. It is an optional attribute and the default is 0. No other synchronous replicas are placed for a partition after a domain reaches this number of synchronous replicas for that specific partition. Adding containers that can support this ObjectGrid can result in an increased number of synchronous replicas if your maxSyncReplicas value has not already been met. The sample set the maxSyncReplicas to 1 means the domain will at most place one synchronous replica. If you start more than one container server instance, there will be only one synchronous replica placed in one of the container server instances.

Using ObjectGrid

The `Client.java` file in the *installRoot*`/ObjectGrid/ gettingstarted/src/` directory is the client program that demonstrates how to connect to catalog server, obtain ObjectGrid instance, and use ObjectMap API.

From the perspective of a client application, using WebSphere eXtreme Scale can be divided into the following steps.

1. Connecting to the catalog service by obtaining a ClientClusterContext instance.
2. Obtaining a client ObjectGrid instance.
3. Getting a Session instance.
4. Getting an ObjectMap instance.
5. Using the ObjectMap methods.

1. Connecting to the catalog service by obtaining a ClientClusterContext instance

To connect to the catalog server, use the connect method of ObjectGridManager API. The connect method that is used by this sample only requires catalog server endpoint in the format of `hostname:port`, such as `localhost:2809`. If the connection to the catalog server succeeds, the connect method returns a ClientClusterContext instance. The ClientClusterContext instance is required to obtain the ObjectGrid from ObjectGridManager API. The following code snippet demonstrates how to connect to a catalog server and obtain a ClientClusterContext instance.

```
ClientClusterContext ccc = ObjectGridManagerFactory.getObjectGridManager().connect("localhost:2809", null, null);
```

2. Obtaining an ObjectGrid instance

To obtain ObjectGrid instance, use the getObjectGrid method of the ObjectGridManager API. The getObjectGrid method requires both the ClientClusterContext instance and the name of the ObjectGrid instance. The ClientClusterContext instance is obtained during connecting to catalog server. The name of ObjectGrid is “Grid” that is specified in the `objectgrid.xml` file. The following code snippet demonstrates how to obtain ObjectGrid by calling getObjectGrid method of ObjectGridManager API.

```
ObjectGrid grid = ObjectGridManagerFactory.getObjectGridManager().getObjectGrid(ccc, "Grid");
```

3. Getting a Session instance

You can get a Session from the obtained ObjectGrid instance. A Session instance is required to get ObjectMap, and perform transaction demarcation. The following code snippet demonstrates how to get Session by calling getSession method of ObjectGrid API.

```
Session sess = grid.getSession();
```

4. Getting an ObjectMap instance

After getting a Session, you can get ObjectMap from a Session by calling getMap method of Session API. You have to pass the name of map as parameter to getMap method in order to get the ObjectMap. The following code snippet demonstrates how to obtain ObjectMap by calling getMap method of Session API.

```
ObjectMap map1 = sess.getMap("Map1");
```

5. Using the ObjectMap methods

After an ObjectMap is obtained, you can use ObjectMap API. Remember ObjectMap is a transactional map and requires transaction demarcation by using the begin and commit methods of the Session API. If there is no explicit transaction demarcation, ObjectMap operations run with auto-commit transactions.

The following code snippet demonstrates how to use ObjectMap API with auto-commit transaction.

```
map1.insert(key1, value1);
```

The following code snippet demonstrates how to use ObjectMap API with explicit transaction demarcation.

```
sess.begin();
map1.insert(key1, value1);
sess.commit();
```

Additional information

This sample demonstrates how to start catalog server and container server and using ObjectMap API in stand-alone environment. You can also use the EntityManager API.

In a WebSphere Application Server environment with WebSphere eXtreme Scale installed or enabled, the most common scenario is a network-attached topology. In a network-attached topology, the catalog server is hosted in the WebSphere Application Server deployment manager process and each WebSphere Application Server instance hosts an eXtreme Scale server automatically. Java Platform, Enterprise Edition applications only need to include both the ObjectGrid descriptor XML file and the ObjectGrid deployment policy descriptor XML file in the `META-INF` directory of any module and the ObjectGrid becomes available automatically. An application can then connect to a locally available catalog server and obtain an ObjectGrid instance to use.

Chapter 2. Programming WebSphere eXtreme Scale

WebSphere eXtreme Scale provides several features that are accessed programmatically using the Java programming language through application programming interfaces (APIs) and system programming interfaces.

WebSphere eXtreme Scale APIs

When you are using eXtreme Scale APIs, you must distinguish between transactional and non-transactional operations. A transactional operation is an operation that is performed within a transaction. ObjectMap, EntityManager, Query, and DataGrid API are transactional APIs that are contained inside the Session that is a transactional container. Non-transactional operations have nothing to do with a transaction, such as configuration operations.

The ObjectGrid, BackingMap, and plug-in APIs are non-transactional. The ObjectGrid, BackingMap, and other configuration APIs are categorized as ObjectGrid Core API. Plug-ins are for customizing the cache to achieve the functions that you want, and are categorized as the System Programming API. A plug-in in eXtreme Scale is a component that provides a certain type of function to the pluggable eXtreme Scale components that include ObjectGrid and BackingMap. A feature represents a specific function or characteristic of an eXtreme Scale component, including ObjectGrid, Session, BackingMap, ObjectMap, and so on. Typically, features are configurable with configuration APIs. Plug-ins can be built-in, but might require that you develop your own plug-ins in some situations.

You can normally configure the ObjectGrid and BackingMap to meet your application requirements. When the application has special requirements, consider using specialized plug-ins. WebSphere eXtreme Scale might have built-in plug-ins that meet your requirements. For example, if you need a peer-to-peer replication model between two local ObjectGrid instances or two distributed eXtreme Scale grids, the built-in JMSObjectGridEventListener is available. If none of the built-in plug-ins can solve your business problems, refer to the System Programming API to provide your own plug-ins.

ObjectMap is a simple map-based API. If the cached objects are simple and no relationship is involved, the ObjectMap API is ideal for your application. If object relationships are involved, use the EntityManager API, which supports graph-like relationships.

Query is a powerful mechanism for finding data in the ObjectGrid. Both Session and EntityManager provide the traditional query capability.

The DataGrid API is a powerful computing capability in a distributed eXtreme Scale environment that involves many machines, replicas, and partitions. Applications can run business logic in parallel in all of the nodes in the distributed eXtreme Scale environment. The application can obtain the DataGrid API through the ObjectMap API.

7.0.0.0 FIX 2+ The WebSphere eXtreme Scale REST data service is a Java HTTP service that is compatible with Microsoft WCF Data Services (formally ADO.NET Data Services) and implements the Open Data Protocol (OData). The REST data service allows any HTTP client to access an eXtreme Scale grid. It is compatible with the WCF Data Services support that is supplied with the Microsoft .NET Framework 3.5 SP1. RESTful applications can be developed with the rich tooling provided by Microsoft Visual Studio 2008 SP1. For more details, refer to the eXtreme Scale REST data service user guide.

Chapter 3. Accessing data in WebSphere eXtreme Scale

After an application has a reference to an ObjectGrid instance or a client connection to a remote grid, you can access and interact with data in your WebSphere eXtreme Scale configuration. With the ObjectGridManager API, use one of the createObjectGrid methods to create a local instance, or the getObjectGrid method for a client instance with a distributed grid.

A thread in an application needs its own Session. When an application wants to use the ObjectGrid on a thread, it should just call one of the getSession methods to obtain a thread. This operation is cheap--there is no need to pool these operations in most cases. If the application is using a dependency injection framework such as Spring, you can inject a Session into an application bean when necessary.

After you obtain a Session, the application can access data stored in maps in the ObjectGrid. If the ObjectGrid uses entities, you can use the EntityManager API, which you can obtain with the Session.getEntityManager method. Because it is closer to Java specifications, the EntityManager interface is simpler than the map-based API. However, the EntityManager API carries a performance overhead because it tracks changes in objects. The map-based API is obtained by using the Session.getMap method.

WebSphere eXtreme Scale uses transactions. When an application interacts with a Session, it must be in the context of a transaction. A transaction is begun and committed or rolled back using the Session.begin, Session.commit, and Session.rollback methods on the Session object. Applications can also work in auto-commit mode, where the Session automatically begins and commits a transaction whenever the application interacts with Maps. However, the auto-commit mode is slower.

The logic of using transactions

Transactions may seem to be slow, but eXtreme Scale uses transactions for three reasons:

1. To allow rollback of changes if an exception occurs or business logic needs to undo state changes.
2. To hold locks on data and release locks within the lifetime of a transaction, allowing a set of changes to be made atomically, that is, all changes or no changes to data.
3. To produce an atomic unit of replication.

WebSphere eXtreme Scale lets a Session customize how much transaction is really needed. An application can turn off rollback support and locking but does so at a cost to the application. The application must handle the lack of these features itself.

For example, an application can turn off locking by configuring the BackingMap locking strategy to be NONE. This strategy is fast, but concurrent transactions can now modify the same data with no protection from each other. The application is responsible for all locking and data consistency when NONE is used.

An application can also change the way objects are copied when accessed by the transaction . The application can specify how objects are copied with the

ObjectMap.setCopyMode method. With this method, you can turn off CopyMode. Turning off CopyMode is normally used for read-only transactions if different values can be returned for the same object within a transaction. Different values can be returned for the same object within a transaction.

For example, if the transaction called the ObjectMap.get method for the object at T1, it got the value at that point in time. If it calls the get method again within that transaction at a later time T2, another thread might have changed the value. Because the value has been changed by another thread, the application sees a different value. If the application modifies an object retrieved using a NONE CopyMode value, it is changing the committed copy of that object directly. Rolling back the transaction has no meaning in this mode. You are changing the only copy in the ObjectGrid. Although using the NONE CopyMode is fast, be aware of its consequences. An application that uses a NONE CopyMode must never roll back the transaction. If the application rolls back the transaction, the indexes are not updated with the changes *and* the changes are not replicated if replication is turned on. The default values are easy to use and less prone to errors. If you start trading performance in exchange for less reliable data, the application needs to be aware of what it is doing to avoid unintended problems.

CAUTION:
Be careful when you are changing either the locking or the CopyMode values. If you change the values, unpredictable application behavior will occur.

Interacting with stored data

After a session has been obtained, you can use the following code fragment to use the Map API for inserting data.

```
Session session = ...;
ObjectMap personMap = session.getMap("PERSON");
session.begin();
Person p = new Person();
p.name = "John Doe";
personMap.insert(p.name, p);
session.commit();
```

The same example using the EntityManager API follows. This code sample assumes that the Person object is mapped to an Entity.

```
Session session = ...;
EntityManager em = session.getEntityManager();
session.begin();
Person p = new Person();
p.name = "John Doe";
em.persist(p);
session.commit();
```

The pattern is designed to obtain references to the ObjectMaps for the Maps that the thread will work with, start a transaction, work with the data, then commit the transaction.

The ObjectMap interface has the typical Map operations such as put, get and remove. However, use the more specific operation names such as: get, getForUpdate, insert, update and remove. These method names convey the intent more precisely that the traditional Map APIs.

You can also use the indexing support, which is flexible.

The following is an example for updating an Object:

```
session.begin();
Person p = (Person)personMap.getForUpdate("John Doe");
p.name = "John Doe";
p.age = 30;
personMap.update(p.name, p);
session.commit();
```

The application normally uses the getForUpdate method rather than a simple get to lock the record. The update method must be called to actually provide the updated value to the Map. If update is not called then the Map is unchanged. The following is the same fragment using the EntityManager API:

```
session.begin();
Person p = (Person)em.findForUpdate(Person.class, "John Doe");
p.age = 30;
session.commit();
```

The EntityManager API is simpler than the Map approach. In this case, eXtreme Scale finds the Entity and returns a managed object to the application. The application modifies the object and commits the transaction, and eXtreme Scale tracks changes to managed objects automatically at commit time and performs the necessary updates.

Transactions and partitions

WebSphere eXtreme Scale transactions can only update a single partition. Transactions from a client can read from multiple partitions, but they can only update one partition. If an application attempts to update two partitions, then the transaction fails and is rolled back. A transaction that is using an embedded ObjectGrid (grid logic) has no routing capability and can only see data in the local partition. This business logic can always get a second session that is a true client session to access other partitions. However, this transaction would be an independent transaction.

Queries and partitions

If a transaction has already searched for an Entity, the transaction is associated with the partition for that Entity. Any queries that run on a transaction that is associated with an Entity are routed to the associated partition.

If a query is run on a transaction before it is associated with a partition, you must set the partition ID to use for the query. The partition ID is an integer value. The query is then routed to that partition.

Queries only search within a single partition. However, you can use the DataGrid APIs to run the same query in parallel on all partitions or a subset of partitions. Use the DataGrid APIs to find an entry that might be in any partition.

7.0.0.0 FIX 2+ The REST data service allows any HTTP client to access a WebSphere eXtreme Scale grid, and is compatible with WCF Data Services in the Microsoft .NET Framework 3.5 SP1. For more information see the user guide for the eXtreme Scale REST data service

.

Interacting with an ObjectGrid using the ObjectGridManager

The ObjectGridManagerFactory class and the ObjectGridManager interface provide a mechanism to create, access, and cache ObjectGrid instances. The ObjectGridManagerFactory class is a static helper class to access the ObjectGridManager interface, a singleton. The ObjectGridManager interface includes several convenience methods to create instances of an ObjectGrid object. The ObjectGridManager interface also facilitates creation and caching of ObjectGrid instances that can be accessed by several users.

Programming model

Before using eXtreme Scale's functionality as an in-memory data grid, you must create and interact with ObjectGrid instances with methods such as the following.

- createObjectGrid methods
- getObjectGrid methods
- removeObjectGrid methods
- controlling the life cycle of an ObjectGrid

createObjectGrid methods

This topic describes the seven createObjectGrid methods in the ObjectGridManager interface. Each of these methods creates a local instance of an ObjectGrid.

Local in-memory instance

The following code snippet illustrates how to obtain and configure a local ObjectGrid instance with eXtreme Scale.

```
// Obtain a local ObjectGrid reference
        // you can create a new ObjectGrid, or get configured ObjectGrid
        // defined in ObjectGrid xml file
        ObjectGridManager objectGridManager =
     ObjectGridManagerFactory.getObjectGridManager();
        ObjectGrid ivObjectGrid =
     objectGridManager.createObjectGrid("objectgridName");

        // Add a TransactionCallback into ObjectGrid
        HeapTransactionCallback tcb = new HeapTransactionCallback();
        ivObjectGrid.setTransactionCallback(tcb);

        // Define a BackingMap
        // if the BackingMap is configured in ObjectGrid xml
        // file, you can just get it.
        BackingMap ivBackingMap = ivObjectGrid.defineMap("myMap");

        // Add a Loader into BackingMap
        Loader ivLoader = new HeapCacheLoader();
        ivBackingMap.setLoader(ivLoader);

        // initialize ObjectGrid
        ivObjectGrid.initialize();

        // Obtain a session to be used by the current thread.
        // Session can not be shared by multiple threads
        Session ivSession = ivObjectGrid.getSession();

        // Obtaining ObjectMap from ObjectGrid Session
        ObjectMap objectMap = ivSession.getMap("myMap");
```

Default shared configuration

The following code is a simple case of creating an ObjectGrid to share among many users.

```
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
final ObjectGridManager oGridManager=
  ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid employees =
 oGridManager.createObjectGrid("Employees",true);
employees.initialize();
employees.
/*sample continues..*/
```

The preceding Java code snippet creates and caches the Employees ObjectGrid. The Employees ObjectGrid is initialized with the default configuration and is ready for use. The second parameter in the createObjectGrid method is set to true, which instructs the ObjectGridManager to cache the ObjectGrid instance it creates. If this parameter is set to false, the instance is not cached. Every ObjectGrid instance has a name, and the instance can be shared among many clients or users based on that name.

If the objectGrid instance is used in peer-to-peer sharing, the caching must be set to true. For more information on peer-to-peer sharing, see Distributing changes between peer Java Virtual Machines.

XML configuration

WebSphere eXtreme Scale is highly configurable. The previous example demonstrates how to create a simple ObjectGrid without any configuration. This example shows you how to create a pre-configured ObjectdGrid instance that is based on an XML configuration file. You can configure an ObjectGrid instance programmatically or using an XML-based configuration file. You can also configure ObjectGrid using a combination of both approaches. The ObjectGridManager interface allows creation of an ObjectGrid instance based on the XML configuration. The ObjectGridManager interface has several methods that take a URL as an argument. Every XML file that is passed into the ObjectGridManager must be validated against the schema. XML validation can be disabled only when the file is previously validated and no changes have been made to the file since its last validation. Disabling validation saves a small amount of overhead but introduces the possibility of using an invalid XML file. The IBM® Java Developer Kit (JDK) 1.4.2 has support for XML validation. When using a JDK that does not have this support, Apache Xerces might be required to validate the XML.

The following Java code snippet demonstrates how to pass in an XML configuration file to create an ObjectGrid.

```
import java.net.MalformedURLException;
import java.net.URL;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
boolean validateXML = true; // turn XML validation on
boolean cacheInstance = true; // Cache the instance
String objectGridName="Employees"; // Name of Object Grid URL
allObjectGrids = new URL("file:test/myObjectGrid.xml");
final ObjectGridManager oGridManager=
 ObjectGridManagerFactory.getObjectGridManager();
 ObjectGrid employees =
 oGridManager.createObjectGrid(objectGridName, allObjectGrids,
  bvalidateXML, cacheInstance);
```

The XML file can contain configuration information for several ObjectGrids. The previous code snippet specifically returns ObjectGrid Employees, assuming that the Employees configuration is defined in the file. For the XML syntax, see ObjectGrid configuration. Seven createObjectGrid methods exist, and are documented in the following code block.

```
/**
 * A simple factory method to return an instance of an
 * Object Grid. A unique name is assigned.
 * The instance of ObjectGrid is not cached.
 * Users can then use {@link ObjectGrid#setName(String)} to change the
 * ObjectGrid name.
 *
 * @return ObjectGrid an instance of ObjectGrid with a unique name assigned
 * @throws ObjectGridException any error encountered during the
 * ObjectGrid creation
 */
public ObjectGrid createObjectGrid() throws ObjectGridException;

/**
 * A simple factory method to return an instance of an ObjectGrid with the
 * specified name. The instances of ObjectGrid can be cached. If an ObjectGrid
 * with the this name has already been cached, an ObjectGridException
 * will be thrown.
 *
 * @param objectGridName the name of the ObjectGrid to be created.
 * @param cacheInstance true, if the ObjectGrid instance should be cached
 * @return an ObjectGrid instance
 * @this name has already been cached or
 * any error during the ObjectGrid creation.
 */
public ObjectGrid createObjectGrid(String objectGridName, boolean cacheInstance)
  throws ObjectGridException;

/**
 * Create an ObjectGrid instance with the specified ObjectGrid name. The
 * ObjectGrid instance created will be cached.
 * @param objectGridName the Name of the ObjectGrid instance to be created.
 * @return an ObjectGrid instance
 * @throws ObjectGridException if an ObjectGrid with this name has already
 * been cached, or any error encountered during the ObjectGrid creation
 */
public ObjectGrid createObjectGrid(String objectGridName)
  throws ObjectGridException;

/**
 * Create an ObjectGrid instance based on the specified ObjectGrid name and the
 * XML file. The ObjectGrid instance defined in the XML file with the specified
 * ObjectGrid name will be created and returned. If such an ObjectGrid
 * cannot be found in the xml file, an exception will be thrown.
 *
 * This ObjecGrid instance can be cached.
 *
 * If the URL is null, it will be simply ignored. In this case, this method behaves
 * the same as {@link #createObjectGrid(String, boolean)}.
 *
 * @param objectGridName the Name of the ObjectGrid instance to be returned. It
 * must not be null.
 * @param xmlFile a URL to a wellformed xml file based on the ObjectGrid schema.
 * @param enableXmlValidation if true the XML is validated
 * @param cacheInstance a boolean value indicating whether the ObjectGrid
 * instance(s)
 * defined in the XML will be cached or not. If true, the instance(s) will
 * be cached.
 *
 * @throws ObjectGridException if an ObjectGrid with the same name
 * has been previously cached, no ObjectGrid name can be found in the xml file,
 * or any other error during the ObjectGrid creation.
 * @return an ObjectGrid instance
 * @see ObjectGrid
 */
public ObjectGrid createObjectGrid(String objectGridName, final URL xmlFile,
final boolean enableXmlValidation, boolean cacheInstance)
 throws ObjectGridException;

/**
 * Process an XML file and create a List of ObjectGrid objects based
 * upon the file.
 * These ObjecGrid instances can be cached.
 * An ObjectGridException will be thrown when attempting to cache a
 * newly created ObjectGrid
 * that has the same name as an ObjectGrid that has already been cached.
 *
 * @param xmlFile the file that defines an ObjectGrid or multiple
 * ObjectGrids
 * @param enableXmlValidation setting to true will validate the XML
 * file against the schema
 * @param cacheInstances set to true to cache all ObjectGrid instances
 * created based on the file
 * @return an ObjectGrid instance
 * @throws ObjectGridException if attempting to create and cache an
 * ObjectGrid with the same name as
 * an ObjectGrid that has already been cached, or any other error
 * occurred during the
 * ObjectGrid creation
 */
public List createObjectGrids(final URL xmlFile, final boolean enableXmlValidation,
boolean cacheInstances) throws ObjectGridException;

/** Create all ObjectGrids that are found in the XML file. The XML file will be
 * validated against the schema. Each ObjectGrid instance that is created will
 * be cached. An ObjectGridException will be thrown when attempting to cache a
 * newly created ObjectGrid that has the same name as an ObjectGrid that has
 * already been cached.
 * @param xmlFile The XML file to process. ObjectGrids will be created based
 * on what is in the file.
 * @return A List of ObjectGrid instances that have been created.
 * @throws ObjectGridException if an ObjectGrid with the same name as any of
 * those found in the XML has already been cached, or
 * any other error encounterred during ObjectGrid creation.
 */
public List createObjectGrids(final URL xmlFile) throws ObjectGridException;

/**
 * Process the XML file and create a single ObjectGrid instance with the
 * objectGridName specified only if an ObjectGrid with that name is found in
 * the file. If there is no ObjectGrid with this name defined in the XML file,
 * an ObjectGridException
 * will be thrown. The ObjectGrid instance created will be cached.
 * @param objectGridName name of the ObjectGrid to create. This ObjectGrid
 * should be defined in the XML file.
 * @param xmlFile the XML file to process
 * @return A newly created ObjectGrid
 * @throws ObjectGridException if an ObjectGrid with the same name has been
 * previously cached, no ObjectGrid name can be found in the xml file,
 * or any other error during the ObjectGrid creation.
 */
public ObjectGrid createObjectGrid(String objectGridName, URL xmlFile)
  throws ObjectGridException;
```

Client hangs during a getObjectGrid method call

A client might seem to hang when calling the getObjectGrid method on the ObjectGridManager or throw an exception: com.ibm.websphere.projector.MetadataException. The EntityMetadata repository is not available and the timeout threshold is reached. The reason is the client is waiting for the entity metadata on the ObjectGrid server to become available. This error can occur when a container has been started, but the initial number of containers or minimum number of synchronous replicas has not been reached. Examine the deployment policy for the ObjectGrid and verify that the number of active containers is greater than or equal to both the numInitialContainers and minSyncReplicas attributes in the deployment policy descriptor file.

getObjectGrid methods

Use the ObjectGridManager.getObjectGrid methods to retrieve cached instances.

Retrieving a cached instance

Since the Employees ObjectGrid instance was cached by the ObjectGridManager interface, another user can access it with the following code snippet:

```
ObjectGrid myEmployees = oGridManager.getObjectGrid("Employees");
```

The following are the two getObjectGrid methods that return cached ObjectGrid instances:

- **Retrieving all cached instances**

  To obtain all of the ObjectGrid instances that have been previously cached, use the getObjectGrids method, which returns a list of each instance. If no cached instances exist, the method will return null.
- **Retrieving a cached instance by name**

  To obtain a single cached instance of an ObjectGrid, use getObjectGrid(String objectGridName), passing the name of the cached instance into the method. The method either returns the ObjectGrid instance with the specified name or returns null if there is no ObjectGrid instance with that name.

Note: You can also use the getObjectGrid method to connect to a distributed grid. See "Connecting to a distributed ObjectGrid" on page 17 for more information.

removeObjectGrid methods

You can use two different removeObjectGrid methods to remove ObjectGrid instances from the cache.

Remove an ObjectGrid instance

To remove ObjectGrid instances from the cache, use one of the removeObjectGrid methods. The ObjectGridManager does not keep a reference of the instances that are removed. Two remove methods exist. One method takes a boolean parameter. If the boolean parameter is set to `true`, the destroy method is called on the ObjectGrid. The call to the destroy method on the ObjectGrid shuts down the ObjectGrid and frees up any resources the ObjectGrid is using. A description of how to use the two removeObjectGrid methods follows:

```
/**
 * Remove an ObjectGrid from the cache of ObjectGrid instances
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
 *
 * @throws ObjectGridException if an ObjectGrid with the objectGridName
 * was not found in the cache
 */
public void removeObjectGrid(String objectGridName) throws ObjectGridException;

/**
 * Remove an ObjectGrid from the cache of ObjectGrid instances and
 * destroy its associated resources
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
 *
 * @param destroy destroy the objectgrid instance and its associated
 * resources
 *
 * @throws ObjectGridException if an ObjectGrid with the objectGridName
 * was not found in the cache
 */
public void removeObjectGrid(String objectGridName, boolean destroy)
 throws ObjectGridException;
```

Connecting to a distributed ObjectGrid

You can connect to a distributed ObjectGrid with a connection end point for the catalog service. You must have the host name and endpoint port of the catalog server to which you want to connect.

In order to connect to a distributed data grid, you must have configured your server-side environment with a catalog service and container servers.

The getObjectGrid(ClientClusterContext ccc, String objectGridName) method connects to the specified catalog service and returns a client ObjectGrid instance corresponding to a server-side ObjectGrid instance.

The following code snippet is an example of how to connect to a distributed data grid.

```
// Create an ObjectGridManager instance.

ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();

// Obtain a ClientClusterContext by connecting to a catalog
// server based distributed ObjectGrid.  You have to provide
// a connection end point for your catalog server in the format
// of hostName:endPointPort.  The hostName is the machine
// where the catalog server resides, and the endPointPort is
// the catalog server's listening port, whose default is 2809.

ClientClusterContext ccc = ogm.connect("localhost:2809", null, null);

// Obtain a distributed ObjectGrid using ObjectGridManager and providing
// the ClientClusterContext.

ObjectGrid og = ogm.getObjectGrid(ccc, "objectgridName");
```

WebSphere eXtreme Scale client configuration

You can configure an eXtreme Scale client based on your requirements such as the need to override settings.

You can configure an eXtreme Scale client in the following ways:
- XML configuration

- Programmatic configuration
- Spring Framework configuration
- Disabling the near cache

You can override the following plug-ins on a client:

- **ObjectGrid plug-ins**
  - TransactionCallback plug-in
  - ObjectGridEventListener plug-in
- **BackingMap plug-ins**
  - Evictor plug-in
  - MapEventListener plug-in
  - numberOfBuckets attribute
  - ttlEvictorType attribute
  - timeToLive attribute

Configure the client with XML

An ObjectGrid XML file can be used to alter settings on the client side. To change the settings on an eXtreme Scale client, you must create an ObjectGrid XML file that is similar in structure to the file that was used for the eXtreme Scale server.

Assume that the following XML file was paired with a deployment policy XML file, and these files were used to start an eXtreme Scale server.

companyGridServerSide.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="CompanyGrid">
            <bean id="TransactionCallback"
                className="com.company.MyTxCallback" />
            <bean id="ObjectGridEventListener"
                className="com.company.MyOgEventListener" />
            <backingMap name="Customer"
                pluginCollectionRef="customerPlugins" />
            <backingMap name="Item" />
            <backingMap name="OrderLine" numberOfBuckets="1049"
                timeToLive="1600" ttlEvictorType="LAST_ACCESS_TIME" />
            <backingMap name="Order" lockStrategy="PESSIMISTIC"
                pluginCollectionRef="orderPlugins" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="customerPlugins">
            <bean id="Evictor"
                className="com.ibm.websphere.objectGrid.plugins.builtins.LRUEvictor" />
            <bean id="MapEventListener"
                className="com.company.MyMapEventListener" />
        </backingMapPluginCollection>
        <backingMapPluginCollection id="orderPlugins">
            <bean id="MapIndexPlugin"
                className="com.company.MyMapIndexPlugin" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

On an eXtreme Scale server, the ObjectGrid instance named CompanyGrid behaves as defined by the `companyGridServerSide.xml` file. By default, the CompanyGrid client has the same settings as the CompanyGrid instance running on the server. However, some of the settings can be overridden on the client, as follows:

1. Create a client-specific ObjectGrid instance.
2. Copy the ObjectGrid XML file that was used to open the server.

3. Edit the new file to customize for the client side.
   - To set or update any of the attributes on the client, specify a new value or change the existing value.
   - To remove a plug-in from the client, use the empty string as the value for the className attribute.
   - To change an existing plug-in, specify a new value for the className attribute.
   - You can also add any plug-in supported for a client override: TRANSACTION_CALLBACK, OBJECTGRID_EVENT_LISTENER, EVICTOR, MAP_EVENT_LISTENER.
4. Create a client with the newly created client-override XML file.

The following ObjectGrid XML file can be used to specify some of the attributes and plug-ins on the CompanyGrid client.

companyGridClientSide.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">

    <objectGrids>
        <objectGrid name="CompanyGrid">
            <bean id="TransactionCallback"
                className="com.company.MyClientTxCallback" />
            <bean id="ObjectGridEventListener" className="" />
            <backingMap name="Customer" numberOfBuckets="1429"
                pluginCollectionRef="customerPlugins" />
            <backingMap name="Item" />
            <backingMap name="OrderLine" numberOfBuckets="701"
                timeToLive="800" ttlEvictorType="LAST_ACCESS_TIME" />
            <backingMap name="Order" lockStrategy="PESSIMISTIC"
                pluginCollectionRef="orderPlugins" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="customerPlugins">
            <bean id="Evictor"
                className="com.ibm.websphere.objectGrid.plugins.builtins.LRUEvictor" />
            <bean id="MapEventListener" className="" />
        </backingMapPluginCollection>
        <backingMapPluginCollection id="orderPlugins">
            <bean id="MapIndexPlugin"
                className="com.company.MyMapIndexPlugin" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

The `companyGridClientSide.xml` file overrides several attributes and plug-ins on the CompanyGrid client as follows:

- The TransactionCallback on the client is com.company.MyClientTxCallback instead of the server-side setting of com.company.MyTxCallback.
- The client does not have an ObjectGridEventListener plug-in because the className value is the empty string.
- The client sets the numberOfBuckets to 1429 for the Customer backingMap, retains its Evictor plug-in, and removes the MapEventListener plug-in.
- The numberOfBuckets and timeToLive attributes of the OrderLine backingMap have changed
- Although a different lockStrategy attribute is specified, there is no effect because the lockStrategy attribute is not supported for a client override.
-

To create the CompanyGrid client using the `companyGridClientSide.xml` file, pass the ObjectGrid XML file as a URL to one of the connect methods on the ObjectGridManager.

creating the client for XML

```
ObjectGridManager ogManager =
 ObjectGridManagerFactory.ObjectGridManager();
```

```
ClientClusterContext clientClusterContext =
 ogManager.connect("MyServer1.company.com:2809", null, new URL(
               "file:xml/companyGridClientSide.xml"));
```

Configure the client programmatically

You can also override client-side ObjectGrid settings programmatically. Create an ObjectGridConfiguration object that is similar in structure to the server-side ObjectGrid instance. The following code creates a client-side ObjectGrid instance that is functionally equivalent to the client override in the previous section which uses an XML file.

client-side override programmatically

```
ObjectGridConfiguration companyGridConfig = ObjectGridConfigFactory
    .createObjectGridConfiguration("CompanyGrid");
Plugin txCallbackPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.TRANSACTION_CALLBACK, "com.company.MyClientTxCallback");
companyGridConfig.addPlugin(txCallbackPlugin);

Plugin ogEventListenerPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.OBJECTGRID_EVENT_LISTENER, "");
companyGridConfig.addPlugin(ogEventListenerPlugin);

BackingMapConfiguration customerMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("Customer");
customerMapConfig.setNumberOfBuckets(1429);
Plugin evictorPlugin = ObjectGridConfigFactory.createPlugin(PluginType.EVICTOR,
    "com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor");
customerMapConfig.addPlugin(evictorPlugin);

companyGridConfig.addBackingMapConfiguration(customerMapConfig);

BackingMapConfiguration orderLineMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("OrderLine");
orderLineMapConfig.setNumberOfBuckets(701);
orderLineMapConfig.setTimeToLive(800);
orderLineMapConfig.setTtlEvictorType(TTLType.LAST_ACCESS_TIME);

companyGridConfig.addBackingMapConfiguration(orderLineMapConfig);

List ogConfigs = new ArrayList();
ogConfigs.add(companyGridConfig);

Map overrideMap = new HashMap();
overrideMap.put(CatalogServerProperties.DEFAULT_DOMAIN, ogConfigs);

ogManager.setOverrideObjectGridConfigurations(overrideMap);
ClientClusterContext client = ogManager.connect(catalogServerAddresses, null, null);
ObjectGrid companyGrid = ogManager.getObjectGrid(client, objectGridName);
```

The ObjectGridManager instance ogManager checks for overrides only in the ObjectGridConfiguration and BackingMapConfiguration objects that you include in the overrideMap Map. For instance, the previous code overrides the number of buckets on the OrderLine Map. However, the Order map remains unchanged on the client side because no configuration for that map is included.

Configure the client in the Spring Framework

Client-side ObjectGrid settings can also be overridden using the Spring Framework. The following example XML file shows how to build an ObjectGridConfiguration element, and use it to override some client side settings. This example calls the same APIs that are demonstrated in the programmatic configuration. The example is also functionally equivalent to the example in the ObjectGrid XML configuration.

client configuration with Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="companyGrid" factory-bean="manager" factory-method="getObjectGrid"
    singleton="true">
    <constructor-arg type="com.ibm.websphere.objectgrid.ClientClusterContext">
      <ref bean="client" />
    </constructor-arg>
    <constructor-arg type="java.lang.String" value="CompanyGrid" />
</bean>

  <bean id="manager" class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
    factory-method="getObjectGridManager" singleton="true">
    <property name="overrideObjectGridConfigurations">
      <map>
        <entry key="DefaultDomain">
          <list>
            <ref bean="ogConfig" />
          </list>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="ogConfig"
    class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
    factory-method="createObjectGridConfiguration">
    <constructor-arg type="java.lang.String">
      <value>CompanyGrid</value>
    </constructor-arg>
    <property name="plugins">
      <list>
   <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="TRANSACTION_CALLBACK" />
          <constructor-arg type="java.lang.String"
            value="com.company.MyClientTxCallback" />
        </bean>
        <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="OBJECTGRID_EVENT_LISTENER" />
          <constructor-arg type="java.lang.String" value="" />
        </bean>
      </list>
 </property>
      <property name="backingMapConfigurations">
        <list>
     <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
            factory-method="createBackingMapConfiguration">
            <constructor-arg type="java.lang.String" value="Customer" />
              <property name="plugins">
      <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
                  factory-method="createPlugin">
                  <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
     value="EVICTOR" />
   <constructor-arg type="java.lang.String"
                    value="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
                </bean>
              </property>
              <property name="numberOfBuckets" value="1429" />
            </bean>
            <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
              factory-method="createBackingMapConfiguration">
                <constructor-arg type="java.lang.String" value="OrderLine" />
                  <property name="numberOfBuckets" value="701" />
   <property name="timeToLive" value="800" />
   <property name="ttlEvictorType">
                    <value type="com.ibm.websphere.objectgrid.
           TTLType">LAST_ACCESS_TIME</value>
   </property>
            </bean>
          </list>
        </property>
      </bean>
      <bean id="client" factory-bean="manager" factory-method="connect"
        singleton="true">
        <constructor-arg type="java.lang.String">
     <value>localhost:2809</value>
        </constructor-arg>
   <constructor-arg
          type="com.ibm.websphere.objectgrid.security.
      config.ClientSecurityConfiguration">
          <null />
        </constructor-arg>
   <constructor-arg type="java.net.URL">
     <null />
        </constructor-arg>
      </bean>
</beans>
```

After creating the XML file, load the file and build the ObjectGrid with the following code snippet.

```
BeanFactory beanFactory = new XmlBeanFactory(new
  UrlResource("file:test/companyGridSpring.xml"));

ObjectGrid companyGrid = (ObjectGrid) beanFactory.getBean("companyGrid");
```

Read about integrating with the Spring framework for more information.

Disable the client near cache

The near cache is enabled by default when locking is configured as optimistic or none. Clients do not maintain a near cache when the locking setting is configured as pessimistic.

To disable the near cache, you must set the numberOfBuckets attribute to 0 in the client override ObjectGrid descriptor file.

See the information about map entry locking in the *Administration Guide* for more information.

Controlling the life cycle of an ObjectGrid

You can use the ObjectGridManager interface to control the life cycle of an ObjectGrid instance using either a startup bean or a servlet.

Managing life cycle with a startup bean

A startup bean is used to control the life cycle of an ObjectGrid instance. A startup bean loads when an application starts. With a startup bean, code can run whenever an application starts or stops as expected. To create a startup bean, use the home com.ibm.websphere.startupservice.AppStartUpHome interface and use the remote com.ibm.websphere.startupservice.AppStartUp interface. Implement the start and stop methods on the bean. The start method is invoked whenever the application starts up. The stop method is invoked when the application shuts down. The start method is used to create ObjectGrid instances. The stop method is used to remove ObjectGrid instances. A code snippet that demonstrates this ObjectGrid life-cycle management in a startup bean follows:

```
public class MyStartupBean implements javax.ejb.SessionBean {
    private ObjectGridManager objectGridManager;

    /* The methods on the SessionBean interface have been
     * left out of this example for the sake of brevity */

    public boolean start(){
        // Starting the startup bean
        // This method is called when the application starts
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create 2 ObjectGrids and cache these instances
            ObjectGrid bookstoreGrid =
      objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
            ObjectGrid videostoreGrid =
      objectGridManager.createObjectGrid("videostore", true);
            // within the JVM,
            // these ObjectGrids can now be retrieved from the
            //ObjectGridManager using the getObjectGrid(String) method
        } catch (ObjectGridException e) {
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public void stop(){
        // Stopping the startup bean
        // This method is called when the application is stopped
        try {
            // remove the cached ObjectGrids and destroy them
            objectGridManager.removeObjectGrid("bookstore", true);
            objectGridManager.removeObjectGrid("videostore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}
```

After the start method is called, the newly created ObjectGrid instances are retrieved from the ObjectGridManager interface. For example, if a servlet is included in the application, the servlet accesses the eXtreme Scale using the following code snippet:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
ObjectGrid videostoreGrid = objectGridManager.getObjectGrid("videostore");
```

Managing life cycle with a servlet

To manage the life cycle of an ObjectGrid in a servlet, you can use the init method to create an ObjectGrid instance and the destroy method to remove the ObjectGrid instance. If the ObjectGrid instance is cached, it is retrieved and manipulated in the servlet code. Sample code that demonstrates ObjectGrid creation, manipulation, and destruction within a servlet follows:

```
public class MyObjectGridServlet extends HttpServlet implements Servlet {
    private ObjectGridManager objectGridManager;

    public MyObjectGridServlet() {
        super();
    }

    public void init(ServletConfig arg0) throws ServletException {
        super.init();
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create and cache an ObjectGrid named bookstore
            ObjectGrid bookstoreGrid =
      objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
        Session session = bookstoreGrid.getSession();
        ObjectMap bookMap = session.getMap("book");
        // perform operations on the cached ObjectGrid
        // ...
    }

    public void destroy() {
        super.destroy();
        try {
            // remove and destroy the cached bookstore ObjectGrid
            objectGridManager.removeObjectGrid("bookstore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}
```

Accessing the ObjectGrid shard

WebSphere eXtreme Scale achieves high processing rates by moving the logic to where the data is and returning only results back to the client.

Application logic in a client Java virtual machine (JVM) needs to pull data from the server JVM that is holding the data and push it back when the transaction commits. This process slows down the rate the data can be processed. If the application logic was on the same JVM as the shard that is holding the data, then the network latency and marshalling cost is eliminated and can provide a significant performance boost.

Local reference to shard data

The ObjectGrid APIs provide a Session to the server-side method. This session is a direct reference to the data for that shard. No routing logic is on that path. The application logic can work with the data for that shard directly. The session cannot be used to access data in another partition because no routing logic exists.

A Loader plug-in also provides a way to receive an event when a shard becomes a primary partition. An application can implement a Loader and implement the ReplicaPreloadController interface. The check preload status method is only called when a shard becomes a primary. The session provided to that method is a local reference to the shards data. This approach is typically used if a partition primary needs to start some threads or subscribe to a message fabric for partition-related traffic. It might start a thread to listen for messages in a local Map using the getNextKey API.

Collocated client-server optimization

If an application uses the client APIs to access a partition that happens to be collocated with the JVM that contains the client, then the network is avoided but some marshalling still occurs because of current implementation issues. If a partitioned grid is used, then no impact on the performance of the application is made because (N-1)/N number of calls route to a different JVM. If you need local access always with a shard, then use the Loader or ObjectGrid APIs to invoke that logic.

Using Sessions to access data in the grid

Applications can begin and end transactions through the Session interface. The Session interface also provides access to the application-based ObjectMap and JavaMap interfaces.

Each ObjectMap or JavaMap instance is directly tied to a specific Session object. Each thread that wants access to an eXtreme Scale must first obtain a Session from the ObjectGrid object. A Session instance cannot be shared concurrently between threads. WebSphere eXtreme Scale does not use any thread local storage, but platform restrictions might limit the opportunity to pass a Session from one thread to another.

Methods

The following methods are available with the Session interface. See the API documentation for more information about the following methods:

```
public interface Session {
    ObjectMap getMap(String cacheName) throws UndefinedMapException;

    void begin() throws TransactionAlreadyActiveException, TransactionException;

    void beginNoWriteThrough() throws TransactionAlreadyActiveException,
   TransactionException;

    public void commit() throws NoActiveTransactionException, TransactionException;

    public void rollback() throws NoActiveTransactionException, TransactionException;

    public void flush() throws TransactionException;

    TxID getTxID() throws NoActiveTransactionException;

    boolean isWriteThroughEnabled();

    void setTransactionType(String tranType);

    public void processLogSequence(LogSequence logSequence) throws
   NoActiveTransactionException, UndefinedMapException, ObjectGridException;

    ObjectGrid getObjectGrid();

    public void setTransactionTimeout(int timeout);
    public int getTransactionTimeout();
    public boolean transactionTimedOut();

    public boolean isCommitting();
    public boolean isFlushing();

    public void markRollbackOnly(Throwable t) throws NoActiveTransactionException;
    public boolean isMarkedRollbackOnly();
}
```

Get method

An application obtains a Session instance from an ObjectGrid object using the ObjectGrid.getSession method. The following example demonstrates how to obtain a Session instance:

```
ObjectGrid objectGrid = ...; Session sess = objectGrid.getSession();
```

After a Session is obtained, the thread keeps a reference to the session for its own use. Calling the getSession method multiple times returns a new Session object each time.

Transactions and Session methods

A Session can be used to begin, commit, or rollback transactions. Operations against BackingMaps using ObjectMaps and JavaMaps are most efficiently performed within a Session transaction. After a transaction has started, any changes to one or more BackingMaps in that transaction scope are stored in a special transaction cache until the transaction is committed. When a transaction is committed, the pending changes are applied to the BackingMaps and Loaders and become visible to any other clients of that ObjectGrid.

WebSphere eXtreme Scale also supports the ability to automatically commit transactions, also known as auto-commit. If any ObjectMap operations are performed outside of the context of an active transaction, an implicit transaction is started before the operation and the transaction is automatically committed before returning control to the application.

```
Session session = objectGrid.getSession();
ObjectMap objectMap = session.getMap("someMap");
session.begin();
objectMap.insert("key1", "value1");
objectMap.insert("key2", "value2");
session.commit();
objectMap.insert("key3", "value3"); // auto-commit
```

Session.flush method

The Session.flush method only makes sense when a Loader is associated with a BackingMap. The flush method invokes the Loader with the current set of changes in the transaction cache. The Loader applies the changes to the backend. These changes are not committed when the flush is invoked. If a Session transaction is committed after a flush invocation, only updates that happen after the flush invocation are applied to the Loader. If a Session transaction is rolled back after a flush invocation, the flushed changes are discarded with all other pending changes in the transaction. Use the Flush method sparingly because it limits the opportunity for batch operations against a Loader. Following is an example of the usage of the Session.flush method:

```
Session session = objectGrid.getSession();
session.begin();
// make some changes
...
session.flush(); // push these changes to the Loader, but don't commit yet
// make some more changes
...
session.commit();
```

NoWriteThrough method

Some eXtreme Scale maps are backed by a Loader, which provides persistent storage for the data in the map. Sometimes it is useful to commit data just to the eXtreme Scale map and not push data out to the Loader. The Session interface provides the beginNoWriteThough method for this purpose. The beginNoWriteThrough method starts a transaction like the begin method. With the beginNoWriteThrough method, when the transaction is committed, the data is only committed to the eXtreme Scale in-memory map and is not committed to the persistent storage that is provided by the Loader. This method is very useful when performing data preload on the map.

When using a distributed ObjectGrid instance, the beginNoWriteThrough method is useful for making changes to the near cache only, without modifying the far cache on the server. If the data is known to be stale in the near cache, using the beginNoWriteThrough method can allow entries to be invalidated on the near cache without invalidating them on the server as well.

The Session interface also provides the isWriteThroughEnabled method to determine what type of transaction is currently active.

```
Session session = objectGrid.getSession();
session.beginNoWriteThrough();
// make some changes ...
session.commit(); // these changes will not get pushed to the Loader
```

Obtain the TxID object method

The TxID object is an opaque object that identifies the active transaction. Use the TxID object for the following purposes:

- For comparison when you are looking for a particular transaction.
- To store shared data between the TransactionCallback and Loader objects.

See TransactionCallback plug-in and Loaders for additional information about the Object slot feature.

Performance monitoring method

If you are using eXtreme Scale within WebSphere Application Server, it might be necessary to reset the transaction type for performance monitoring. You can set the transaction type with the setTransactionType method. See Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI) for more information about the setTransactionType method.

Process a complete LogSequence method

WebSphere eXtreme Scale can propagate sets of map changes to ObjectGrid listeners as a means of distributing maps from one Java virtual machine to another. To make it easier for the listener to process the received LogSequences, the Session interface provides the processLogSequence method. This method examines each LogElement within the LogSequence and performs the appropriate operation, for example, insert, update, invalidate, and so on, against the BackingMap that is identified by the LogSequence MapName. An ObjectGrid Session must be available before the processLogSequence method is invoked. The application is also responsible for issuing the appropriate commit or rollback calls to complete the Session. Autocommit processing is not available for this method invocation. Normal processing by the receiving ObjectGridEventListener at the remote JVM would be to start a Session using the beginNoWriteThrough method, which prevents endless propagation of changes, followed by a call to this processLogSequence method, and then committing or rolling back the transaction.

```
// Use the Session object that was passed in during
//ObjectGridEventListener.initialization...
session.beginNoWriteThrough();
// process the received LogSequence
try {
 session.processLogSequence(receivedLogSequence);
} catch (Exception e) {
 session.rollback(); throw e;
}
// commit the changes
session.commit();
```

markRollbackOnly method

This method is used to mark the current transaction as "rollback only". Marking a transaction "rollback only" ensures that even if the commit method is called by application, the transaction is rolled back. This method is typically used by ObjectGrid itself or by the application when it knows that data corruption could occur if the transaction was allowed to be committed. After this method is called, the Throwable object that is passed to this method is chained to the com.ibm.websphere.objectgrid.TransactionException exception that results by the commit method if it is called on a Session that was previously marked a "rollback only". Any subsequent calls to this method for a transaction that is already marked as "rollback only" is ignored. That is, only the first call that passes a non-null Throwable reference is used. Once the marked transaction is completed, the "rollback only" mark is removed so that the next transaction that is started by the Session can be committed.

isMarkedRollbackOnly method

Returns if Session is currently marked as "rollback only". Boolean true is returned by this method if and only if markRollbackOnly method was previously called on this Session and the transaction started by the Session is still active.

setTransactionTimeout method

Set transaction timeout for next transaction started by this Session to a specified number of seconds. This method does not affect the transaction timeout of any transactions previously started by this Session. It only affects transactions that are started after this method is called. If this method is never called, then the timeout value that was passed to the setTxTimeout method of the com.ibm.websphere.objectgrid.ObjectGrid method is used.

getTransactionTimeout method

This method returns the transaction timeout value in seconds. The last value that was passed as the timeout value to the setTransactionTimeout method is returned by this method. If the setTransactionTimeout method is never called, then the timeout value that was passed to the setTxTimeout method of the com.ibm.websphere.objectgrid.ObjectGrid method is used.

transactionTimedOut

This method returns boolean true if the current transaction that was started by this Session has timed out.

isFlushing method

This method returns boolean true if and only if all transaction changes are being flushed out to the Loader plug-in as a result of the flush method of Session interface being invoked. A Loader plug-in may find this method useful when it needs to know why its batchUpdate method was invoked.

isCommitting method

This method returns boolean true if and only if all transaction changes are being committed as a result of the commit method of Session interface being invoked. A Loader plug-in might find this method useful when it needs to know why its batchUpdate method was invoked.

setRequestRetryTimeout method

This method sets the request retry timeout value for the session in milliseconds. If the client set a request retry timeout, the session setting overrides the client value.

getRequestRetryTimeout method

This method gets the current request retry timeout setting on the session. A value of `-1` indicates that the timeout is not set. A value of `0` indicates it is in fail-fast mode. A value greater than `0` indicates the timeout setting in milliseconds.

Handling locks

Locks have life cycles and different types of locks are compatible with others in various ways. Locks must be handled in the correct order to avoid deadlock scenarios.

Lock life cycle

Lock time outs

Each BackingMap has a default lock wait timeout value of 15 seconds. The timeout value is used to ensure that an application does not wait endlessly for a lock mode to be granted because of a deadlock condition that occurs due to an application error. Your application can use the BackingMap interface to override the default lock wait timeout value. The following example illustrates how to set the lock wait timeout value for the map1 backing map to 60 seconds:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm.setLockTimeout( 60 );
```

See Constant field values for more information about the default values of the backing map configuration.

To avoid a java.lang.IllegalStateException exception, call both the setLockStrategy method and the setLockTimeout method before calling either the initialize or getSession methods on the ObjectGrid instance. The setLockTimeout method parameter is a Java primitive integer that specifies the number of seconds that eXtreme Scale waits for a lock mode to be granted. If a transaction waits longer than the lock wait timeout value configured for the BackingMap, a com.ibm.websphere.objectgrid.LockTimeoutException exception results.

When a LockTimeoutException occurs, the application must determine if the timeout is occurring because the application is running slower than expected, or if the timeout occurred because of a deadlock condition. If an actual deadlock condition occurred, then increasing the lock wait timeout value does not eliminate the exception. Increasing the timeout results in the exception taking longer to occur. However, if increasing the lock wait timeout value does eliminate the exception, then the problem occurred because the application was running slower than expected. The application in this case must determine why performance is slow.

Lock wait timeout for ObjectMaps

The lock wait timeout can be overridden for a single ObjectMap instance by using the ObjectMap.setLockTimeout method. The lock timeout value affects all transactions started after the new timeout value is set. This method can be useful when lock collisions are possible or expected in select transactions.

Shared, upgradeable, and exclusive locks

When an application calls any method of the ObjectMap interface, uses the find methods on an index, or does a query, eXtreme Scale automatically attempts to acquire a lock for the map entry that is being accessed. WebSphere eXtreme Scale uses the following lock modes based on the method the application calls in the ObjectMap interface.

- The get and getAll methods on the ObjectMap interface, index methods, and queries acquire an *S lock*, or a shared lock mode for the key of a map entry. The duration that the S lock is held depends on the transaction isolation level used. An S lock mode allows concurrency between transactions that attempt to acquire an S or an upgradeable lock (U lock) mode for the same key, but blocks other transactions that attempt to get an exclusive lock (X lock) mode for the same key.
- The getForUpdate and getAllForUpdate methods acquire a *U lock*, or an upgradeable lock mode for the key of a map entry. The U lock is held until the transaction completes. A U lock mode allows concurrency between transactions that acquire an S lock mode for the same key, but blocks other transactions that attempt to acquire a U lock or X lock mode for the same key.
- The put, putAll, remove, removeAll, insert, update, and touch acquire an *X lock*, or exclusive lock mode for the key of a map entry. The X lock is held until the transaction completes. An X lock mode ensures that only one transaction is inserting, updating, or removing a map entry of a given key value. An X lock blocks all other transactions that attempt to acquire a S, U, or X lock mode for the same key.
- The global invalidate and global invalidateAll methods acquire an X lock for each map entry that is invalidated. The X lock is held until the transaction completes. No locks are acquired for the local invalidate and local invalidateAll methods because none of the BackingMap entries are invalidated by local invalidate method calls.

From the preceding definitions, it is obvious that an S lock mode is weaker than a U lock mode because it allows more transactions to run concurrently when accessing the same map entry. The U lock mode is slightly stronger than the S lock mode because it blocks other transactions that are requesting either a U or X lock mode. The S lock mode only blocks other transactions that are requesting an X lock mode. This small difference is important in preventing some deadlocks from occurring. The X lock mode is the strongest lock mode because it blocks all other transactions attempting to get an S, U, or X lock mode for the same map entry. The net effect of an X lock mode is to ensure that only one transaction can insert, update, or remove a map entry and to prevent updates from being lost when more than one transaction is attempting to update the same map entry.

The following table is a lock mode compatibility matrix that summarizes the described lock modes, which you can use to determine which lock modes are compatible with each other. To read this matrix, the row in the matrix indicates a lock mode that is already granted. The column indicates the lock mode that is requested by another transaction. If Yes is displayed in the column, then the lock mode requested by the other transaction is granted because it is compatible with the lock mode that is already granted. No indicates that the lock mode is not compatible and the other transaction must wait for the first transaction to release the lock that it owns.

Table 1. Lock mode compatibility matrix

Lock	Lock type S (shared)	Lock type U (upgradeable)	Lock type X (exclusive)	Strength
S (shared)	Yes	Yes	No	weakest
U (upgradeable)	Yes	No	No	normal
X (exclusive)	No	No	No	strongest

Locking deadlocks

Consider the following sequence of lock mode requests:

1. X lock is granted to transaction 1 for key1.
2. X lock is granted to transaction 2 for key2.
3. X lock requested by transaction 1 for key2. (Transaction 1 blocks waiting for lock owned by transaction 2.)
4. X lock requested by transaction 2 for key1. (Transaction 2 blocks waiting for lock owned by transaction 1.)

The preceding sequence is the classic deadlock example of two transactions that attempt to acquire more than a single lock, and each transaction acquires the locks in a different order. To prevent this deadlock, each transaction must obtain the multiple locks in the same order. If the OPTIMISTIC lock strategy is used and the flush method on the ObjectMap interface is never used by the application, then lock modes are requested by the transaction only during the commit cycle. During the commit cycle, eXtreme Scale determines the keys for the map entries that need to be locked and requests the lock modes in key sequence (deterministic behavior). With this method, eXtreme Scale prevents the large majority of the classic deadlocks. However, eXtreme Scale does not and cannot prevent all possible deadlock scenarios. A few scenarios exist that the application needs to consider. Following are the scenarios that the application must be aware of and take preventative action against.

One scenario exists where eXtreme Scale is able to detect a deadlock without having to wait for a lock wait timeout to occur. If this scenario does occur, a com.ibm.websphere.objectgrid.LockDeadlockException exception results. Consider the following code snippet:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
sess.begin();
Person p = (IPerson)person.get("Lynn");
// Lynn had a birthday, so we make her 1 year older.
p.setAge( p.getAge() + 1 );
person.put( "Lynn", p );
sess.commit();
```

In this situation, Lynn's boyfriend wants to make her older than she is now, and both Lynn and her boyfriend run this transaction concurrently. In this situation, both transactions own an S lock mode on the Lynn entry of the PERSON map as a result of the person.get("Lynn") method invocation. As a result of the person.put ("Lynn", p) method call, both transactions attempt to upgrade the S lock mode to an X lock mode. Both transactions block waiting for the other transaction to release the S lock mode it owns. As a result, a deadlock occurs because a circular wait condition exists between the two transactions. A circular wait condition results when more than one transaction attempts to promote a lock from a weaker to a stronger mode for the same map entry. In this scenario, a LockDeadlockException exception results instead of a LockTimeoutException exception.

The application can prevent the LockDeadlockException exception for the preceding example by using the optimistic lock strategy instead of the pessimistic lock strategy. Using the optimistic lock strategy is the preferred solution when the map is mostly read and updates to the map are infrequent. If the pessimistic lock strategy must be used, the getForUpdate method can be used instead of the get method in the above example or a transaction isolation level of TRANSACTION_READ_COMMITTED can be used.

For more information, see the topic on locking strategies in the *Product Overview.*

Using the TRANSACTION_READ_COMMITTED transaction isolation level prevents the S lock that is acquired by the get method from being held until the transaction completes. If the key is never invalidated in the transactional cache, repeatable reads are still guaranteed.

See the topic on map entry locking in the *Administration Guide* for more information.

An alternative to changing the transaction isolation level is to use the getForUpdate method. The first transaction to call the getForUpdate method acquires a U lock mode instead of an S lock. This lock mode causes the second transaction to block when it calls the getForUpdate method because only one transaction is granted a U lock mode. Because the second transaction is blocked, it does not own any lock mode on the Lynn map entry. The first transaction does not block when it attempts to upgrade the U lock mode to an X lock mode as a result of the put method call from the first transaction. This feature demonstrates why U lock mode is called the *upgradeable* lock mode. When the first transaction is completed, the second transaction unblocks and is granted the U lock mode. An application can prevent the lock promotion deadlock scenario by using the getForUpdate method instead of the get method when pessimistic lock strategy is being used.

Important: This solution does not prevent read-only transactions from being able to read a map entry. Read-only transactions call the get method, but never call the put, insert, update, or remove methods. Concurrency is just as high as when the regular get method is used. The only reduction in concurrency occurs when the getForUpdate method is called by more than one transaction for the same map entry.

You must be aware when a transaction calls the getForUpdate method on more than one map entry to ensure that the U locks are acquired in the same order by each transaction. For example, suppose that the first transaction calls the getForUpdate method for the key 1 and the getForUpdate method for key 2. Another concurrent transaction calls the getForUpdate method for the same keys, but in reverse order. This sequence causes the classic deadlock because multiple locks are obtained in different orders by different transactions. The application still needs to ensure that every transaction accesses multiple map entries in key sequence to ensure that deadlock does not occur. Because the U lock is obtained at the time that the getForUpdate method is called rather than at commit time, the eXtreme Scale cannot order the lock requests like it does during the commit cycle. The application must control the lock ordering in this case.

Using the flush method on the ObjectMap interface before a commit can introduce additional lock ordering considerations. The flush method is typically used to force changes made to the map out to the backend through the Loader plug-in. In this situation, the backend uses its own lock manager to control concurrency, so the lock wait condition and deadlock can occur in backend rather than in the eXtreme Scale lock manager. Consider the following transaction:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    person.flush();
    ...
    p = (IPerson)person.get("Tom");
    p.setAge( p.getAge() + 1 );
    sess.commit();
    activeTran = false;
}
finally
{
    if ( activeTran ) sess.rollback();
}
```

Suppose that another transaction also updated the Tom person, called the flush method, and then updated the Lynn person. If this situation occurred, the following interleaving of the two transactions results in a database deadlock condition:

```
X lock is granted to transaction 1 for "Lynn" when flush is executed.
X lock is granted to transaction 2 for "Tom" when flush is executed..
X lock requested by transaction 1 for "Tom" during commit processing.
(Transaction 1 blocks waiting for lock owned by transaction 2.)
X lock requested by transaction 2 for "Lynn" during commit processing.
(Transaction 2 blocks waiting for lock owned by transaction 1.)
```

This example demonstrates that the use of the flush method can cause a deadlock to occur in the database rather than in eXtreme Scale. This deadlock example can occur regardless of what lock strategy is used. The application must take care to prevent this kind of deadlock from occurring when using the flush method and when a Loader is plugged into the BackingMap. The preceding example also illustrates another reason why eXtreme Scale has a lock wait timeout mechanism. A transaction that is waiting for a database lock might be waiting while it owns an eXtreme Scale map entry lock. Consequently, problems at database level can cause excessive wait times for an eXtreme Scale lock mode and result in a LockTimeoutException exception.

Common deadlock scenarios

The following sections describe some of the most common deadlock scenarios and suggestions on how to avoid them.

Scenario: Single key deadlocks

The following scenarios describe how deadlocks can occur when a single key is accessed using a S lock and later updated. When this happens from two transactions simultaneously, it results in a deadlock.

Table 2. Single key deadlocks scenario

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)	map.get(key1)	S lock granted to both transactions for key1.
3	map.update(Key1,v)		No U lock. Update performed in transactional cache.
4		map.update(key1,v)	No U lock. Update performed in the transactional cache
5	session.commit()		Blocked: The S lock for key1 cannot be upgraded to an X lock because Thread 2 has an S lock.
6		session.commit()	Deadlock: The S lock for key1 cannot be upgraded to an X lock because T1 has an S lock.

Table 3. Single key deadlocks, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)		S lock granted for key1
3	map.getForUpdate(key1,v)		S lock is upgraded to a U lock for key1.
4		map.get(key1)	S lock granted for key1.
5		map.getForUpdate(key1,v)	Blocked: T1 already has U lock.
6	session.commit()		Deadlock: The U lock for key1 cannot be upgraded.
7		session.commit()	Deadlock: The S lock for key1 cannot be upgraded.

Table 4. Single key deadlocks, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establish an independent transaction
2	map.get(key1)		S lock granted for key1.
3	map.getForUpdate(key1,v)		S lock is upgraded to a U lock for key1
4		map.get(key1)	S lock is granted for key1.
5		map.getForUpdate(key1,v)	Blocked: Thread 1 already has a U lock.
6	session.commit()		Deadlock: The U lock for key1 cannot be upgraded to an X lock because Thread 2 has an S lock.

If the ObjectMap.getForUpdate is used to avoid the S lock, then the deadlock is avoided:

Table 5. Single key deadlocks, continued

	Thread 1	**Thread 2**	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getForUpdate(key1)		U lock granted to thread 1 for key1.
3		map.getForUpdate(key1)	U lock request is blocked.
4	map.update(key1,v)	<blocked>	
5	session.commit()	<blocked>	The U lock for key1 can be successfully upgraded to an X lock.
6		<released>	The U lock is finally granted to key1 for thread 2.
7		map.update(key2,v)	U lock granted to thread 2 for key2.
8		session.commit()	The U lock for key1 can successfully be upgraded to an X lock.

Solutions

1. Use the getForUpdate method instead of get to acquire a U lock instead of an S lock.
2. Use a transaction isolation level of read committed to avoid holding S locks. Reducing the transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads are only possible if the transaction cache is explicitly invalidated.
3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.

Scenario: Ordered multiple key deadlock

This scenario describes what happens if two transactions attempt to update the same entry directly and hold S locks to other entries.

Table 6. Ordered multiple key deadlock scenario

	Thread 1	**Thread 2**	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)	map.get(key1)	S lock granted to both transactions for key1.
3	map.get(key2)	map.get(key2)	S lock granted to both transactions for key2.
4	map.update(key1,v)		No U lock. Update performed in transactional cache.
5		map.update(key2,v)	No U lock. Update performed in transactional cache.
6.	session.commit()		Blocked: The S lock for key 1 cannot be upgraded to an X lock because thread 2 has an S lock.

Table 6. Ordered multiple key deadlock scenario (continued)

	Thread 1	Thread 2	
7		session.commit()	Deadlock: The S lock for key 2 cannot be upgraded because thread 1 has an S lock.

You can use the ObjectMap.getForUpdate method to avoid the S lock, then you can avoid the deadlock:

Table 7. Ordered multiple key deadlock scenario, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getForUpdate(key1)		U lock granted to transaction T1 for key1.
3		map.getForUpdate(key1)	U lock request is blocked.
4	map.get(key2)	<blocked>	S lock granted for T1 for key2.
5	map.update(key1,v)	<blocked>	
6	session.commit()	<blocked>	The U lock for key1 can be successfully upgraded to an X lock.
7		<released>	The U lock is finally granted to key1 for T2
8		map.get(key2)	S lock granted to T2 for key2.
9		map.update(key2,v)	U lock granted to T2 for key2.
10		session.commit()	The U lock for key1 can be successfully upgraded to an X lock.

Solutions

1. Use getForUpdate instead of the get method to acquire a U lock directly for the first key. This strategy works only if the method order is deterministic.
2. Use a transaction isolation level of read committed to avoid holding S locks. This solution is the easiest to implement if the method order is not deterministic. Reducing the transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads are only possible if the transaction cache is explicitly invalidated.
3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.

Scenario: Out of order with U lock

If the order in which keys are requested cannot be guaranteed, then a deadlock can still occur:

Table 8. Out of order with U lock scenario

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getforUpdate(key1)	map.getForUpdate(key2)	U locks successfully granted for key1 and key2.
3	map.get(key2)	map.get(key1)	S lock granted for key1 and key2.
4	map.update(key1,v)	map.update(key2,v)	

Table 8. Out of order with U lock scenario (continued)

	Thread 1	**Thread 2**	
5	session.commit()		The U lock cannot be upgraded to an X lock because T2 has an S lock.
6		session.commit()	The U lock cannot be upgraded to an X lock because T1 has an S lock.

Solutions

1. Wrap all work with a single global U lock (mutex). This method reduces concurrency, but handles all scenarios when access and order is non-deterministic.
2. Use a transaction isolation level of read committed to avoid holding S locks. Reducing the transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads from one client are only possible if the transaction cache is explicitly invalidated by the same client.
3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.

Exception handling in locking scenarios

The preceding examples do not have any exception handling. To prevent locks from being held for excessive amounts of time when a LockTimeoutException exception or a LockDeadlockException exception occurs, an application must ensure that it catches unexpected exceptions and calls the rollback method when something unexpected occurs. Change the preceding code snippet as demonstrated in the following example:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    // Lynn had a birthday, so we make her 1 year older.
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    sess.commit();
    activeTran = false;
}
finally
{
    if ( activeTran ) sess.rollback();
}
```

The finally block in the snippet of code ensures that a transaction is rolled back when an unexpected exception occurs. It not only handles a LockDeadlockException exception, but any other unexpected exception that might occur. The finally block handles the case where an exception occurs during a commit method invocation. This example is not the only way to deal with unexpected exceptions, and there might be cases where an application wants to catch some of the unexpected exceptions that can occur and display one of its application exceptions. You can add catch blocks as appropriate, but the application must ensure that the snippet of code does not exit without completing the transaction.

Transaction isolation

For transactions, you can configure each backing map configuration with one of three lock strategies: pessimistic, optimistic or none. When you are using pessimistic and optimistic locking, eXtreme Scale uses shared (S), upgradeable (U) and exclusive (X) locks to maintain consistency. This locking behavior is most notable when using pessimistic locking, because optimistic locks are not held. You can use one of three transaction isolation levels to tune the locking semantics that eXtreme Scale uses to maintain consistency in each cache map: repeatable read, read committed and read uncommitted.

Transaction isolation overview

Transaction isolation defines how the changes that are made by one operation become visible to other concurrent operations.

WebSphere eXtreme Scale supports three transaction isolation levels with which you can further tune the locking semantics that eXtreme Scale uses to maintain consistency in each cache map: repeatable read, read committed and read uncommitted. The transaction isolation level is set on the Session interface using the setTransactionIsolation method. The transaction isolation can be changed any time during the life of the session, if a transaction is not currently in progress.

The product enforces the various transaction isolation semantics by adjusting the way in which shared (S) locks are requested and held. Transaction isolation has no effect on maps configured to use the optimistic or none locking strategies or when upgradeable (U) locks are acquired.

Repeatable read with pessimistic locking

The repeatable read transaction isolation level is the default. This isolation level prevents dirty reads and non-repeatable reads, but does not prevent phantom reads. A dirty read is a read operation that occurs on data that has been modified by a transaction but has not been committed. A non-repeatable read might occur when read locks are not acquired when performing a read operation. A phantom read can occur when two identical read operations are performed, but two different sets of results are returned because an update has occurred on the data between the read operations. The product achieve a repeatable read by holding onto any S locks until the transaction that owns the lock completes. Because an X lock is not granted until all S locks are released, all transactions holding the S lock are guaranteed to see the same value when re-read.

```
map = session.getMap("Order");
session.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session.begin();

// An S lock is requested and held and the value is copied into
// the transactional cache.
Order order = (Order) map.get("100");
// The entry is evicted from the transactional cache.
map.invalidate("100", false);

// The same value is requested again.  It already holds the
// lock, so the same value is retrieved and copied into the
// transactional cache.
Order order2 (Order) = map.get("100");

// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

Phantom reads are possible when you are using queries or indexes because locks are not acquired for ranges of data, only for the cache entries that match the index or query criteria. For example:

```
session1.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session1.begin();

// A query is run which selects a range of values.
ObjectQuery query = session1.createObjectQuery
    ("SELECT o FROM Order o WHERE o.itemName='Widget'");

// In this case, only one order matches the query filter.
// The order has a key of "100".
// The query engine automatically acquires an S lock for Order "100".
Iterator result = query.getResultIterator();

// A second transaction inserts an order that also matches the query.
Map orderMap = session2.getMap("Order");
orderMap.insert("101", new Order("101", "Widget"));

// When the query runs again in the current transaction, the
// new order is visible and will return both Orders "100" and "101".
result = query.getResultIterator();

// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

Read committed with pessimistic locking

The read committed transaction isolation level can be used with eXtreme Scale, which prevents dirty reads, but does not prevent non-repeatable reads or phantom reads, so eXtreme Scale continues to use S locks to read data from the cache map, but immediately releases the locks.

```
map1 = session1.getMap("Order");
session1.setTransactionIsolation(Session.TRANSACTION_READ_COMMITTED);
session1.begin();

// An S lock is requested but immediately released and
//the value is copied into the transactional cache.

Order order = (Order) map1.get("100");

// The entry is evicted from the transactional cache.
map1.invalidate("100", false);

// A second transaction updates the same order.
// It acquires a U lock, updates the value, and commits.
// The ObjectGrid successfully acquires the X lock during
// commit since the first transaction is using read
// committed isolation.

Map orderMap2 = session2.getMap("Order");
session2.begin();
order2 = (Order) orderMap2.getForUpdate("100");
order2.quantity=2;
orderMap2.update("100", order2);
session2.commit();

// The same value is requested again.  This time, they
// want to update the value, but it now reflects
// the new value
Order order1Copy (Order) = map1.getForUpdate("100");
```

Read uncommitted with pessimistic locking

The read uncommitted transaction isolation level can be used with eXtreme Scale, which is a level that allows dirty reads, non-repeatable reads and phantom reads.

SessionHandle for routing

When using a per-container partition placement policy, you can use a SessionHandle. A SessionHandle instance contains partition information for the current Session and can be reused for a new Session.

A SessionHandle includes information for the partition to which the current Session is bound. SessionHandle is extremely useful for the per-container partition placement policy and can be serialized with standard Java serialization.

If you have a SessionHandle instance, you can apply that handle to a Session with the setSessionHandle(SessionHandle target) method, passing the handle in as the target. You can retrieve the SessionHandle with the Session.getSessionHandle method.

Because it is only applicable in a per-container placement scenario, getting the SessionHandle throws an IllegalStateException if a given ObjectGrid has multiple per-container mapsets or has none. If you do not invoke the setSessionHandle method before calling the getSessionHandle method, the appropriate SessionHandle will be selected based on your ClientProperties configuration.

You can also use the helper class SessionHandleTransformer to convert the handle into different formats. The methods of this class can change a handle's representation from byte array to instance, string to instance, and vice versa for both cases, and can also write the handle's contents into the output stream.

For an example on how you can use a SessionHandle, see the zone-preferred routing topic in the *Product Overview*.

Optimistic collision exception

You can receive an OptimisticCollisionException directly, or receive it with an ObjectGridException.

The following code is an example of how to catch the exception and then display its message:

```
try {
...
} catch (ObjectGridException oe) {
    System.out.println(oe);
}
```

Exception cause

OptimisticCollisionException is created in a situation in which two different clients try to update the same map entry at relatively the same time. For example, if one client attempts to commit a session and update the map entry after another client reads the data before the commit, that data is then incorrect. The exception is created when the other client attempts to commit the incorrect data.

Retrieving the key that triggered the exception

It might be useful, when troubleshooting such an exception, to retrieve the key corresponding to the entry that triggered the exception. The benefit of the OptimisticCollisionException is it contains the getKey method, which returns the object representing that key. The following example demonstrates how to retrieve and print the key when catching OptimisticCollisionException:

```
try {
...
} catch (OptimisticCollisionException oce) {
    System.out.println(oce.getKey());
}
```

ObjectGridException causes an OptimisticCollisionException

OptimisticCollisionException might be the cause of ObjectGridException displaying. If this is the case, you can use the following code to determine the exception type and print out the key. The following code uses the findRootCause utility method as described in the section below.

```
try {
...
}
catch (ObjectGridException oe) {
    Throwable Root = findRootCause( oe );
    if (Root instanceof OptimisticCollisionException) {
        OptimisticCollisionException oce = (OptimisticCollisionException)Root;
        System.out.println(oce.getKey());
    }
}
```

General exception handling technique

Knowing the root cause of a Throwable object is helpful in isolating the source of an error. The following example demonstrates how an exception handler uses a utility method to find the root cause of the Throwable object.

Example:

```
static public Throwable findRootCause( Throwable t )
{
     // Start with Throwable that occurred as the root.
     Throwable root = t;

     // Follow cause chain until last Throwable in chain is found.
     Throwable cause = root.getCause();
     while ( cause != null )
     {
        root = cause;
        cause = root.getCause();
     }

     // Return last Throwable in the chain as the root cause.
     return root;
}
```

ObjectMap API

ObjectMaps are like Java Maps that allow data to be stored as key-value pairs. ObjectMaps provide a simple and intuitive approach for the application to store data. An ObjectMap is ideal for caching objects that have no relationships involved. If object relationships are involved, then you should use the EntityManager API.

For more information about the EntityManager API, see “Caching objects and their relationships (EntityManager API)” on page 51.

Applications typically obtain a WebSphere eXtreme Scale reference and then obtain a Session object from the reference for each thread. Sessions cannot be shared between threads. The getMap method of Session returns a reference to an ObjectMap to use for this thread.

Introduction to ObjectMap

The ObjectMap interface is used for transactional interaction between applications and BackingMaps.

Purpose

An ObjectMap instance is obtained from a Session object that corresponds to the current thread. The ObjectMap interface is the main vehicle that applications use to make changes to entries in a BackingMap.

Obtain an ObjectMap instance

An application gets an ObjectMap instance from a Session object using the Session.getMap(String) method. The following code snippet demonstrates how to obtain an ObjectMap instance:

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
```

Each ObjectMap instance corresponds to a particular Session object. Calling the getMap method multiple times on a particular Session object with the same BackingMap name always returns the same ObjectMap instance.

Automatically commit transactions

Operations against BackingMaps that use ObjectMaps and JavaMaps are performed most efficiently within a Session transaction. WebSphere eXtreme Scale provides autocommit support when methods on the ObjectMap and JavaMap interfaces are called outside of a Session transaction. The methods start an implicit transaction, perform the requested operation, and commit the implicit transaction.

Method semantics

An explanation of the semantics behind each method on the ObjectMap and JavaMap interfaces follows. The setDefaultKeyword method, the invalidateUsingKeyword method, and the methods that have a Serializable argument are discussed in the Keywords topic. The setTimeToLive method is discussed in the Evictors topic. See the API documentation for more information on these methods.

containsKey method

The containsKey method determines if a key has a value in the BackingMap or Loader. If null values are supported by an application, this method can be used to determine if a null reference that is returned from a get operation refers to a null value or indicates that the BackingMap and Loader do not contain the key.

flush method

The flush method semantics are similar to the flush method on the Session interface. The notable difference is that the Session flush applies the current pending changes for all of the maps that are modified in the current session. With this method, only the changes in this ObjectMap instance are flushed to the loader.

get method

The get method fetches the entry from the BackingMap instance. If the entry is not found in the BackingMap instance but a Loader is associated with the BackingMap instance, the BackingMap instance attempts to fetch the entry from the Loader. The getAll method is provided to allow batch fetch processing.

getForUpdate method

The getForUpdate method is the same as the get method, but using the getForUpdate method tells the BackingMap and Loader that the intention is to update the entry. A Loader can use this hint to issue a SELECT for UPDATE query to a database backend. If a pessimistic locking strategy is defined for the BackingMap, the lock manager locks the entry. The getAllForUpdate method is provided to allow batch fetch processing.

insert method

The insert method inserts an entry into the BackingMap and the Loader. Using this method tells the BackingMap and Loader that you want to insert an entry that did not previously exist. When you invoke this method on an existing entry, an exception occurs when the method is invoked or when the current transaction is committed.

invalidate method

The semantics of the invalidate method depend on the value of the **`isGlobal`** parameter that is passed to the method. The invalidateAll method is provided to allow batch invalidate processing.

Local invalidation is specified when the value false is passed as the **`isGlobal`** parameter of the invalidate method. Local invalidation discards any changes to the entry in the transaction cache. If the application issues a get method, the entry is fetched from the last committed value in the BackingMap. If no entry is present in the BackingMap, the entry is fetched from the last flushed or committed value in the Loader. When a transaction is committed, any entries that are marked as locally invalidated have no impact on the BackingMap. Any changes that were flushed to the Loader are still committed even if the entry was invalidated.

Global invalidation is specified when true is passed as the **`isGlobal`** parameter of the invalidate method. Global invalidation discards any pending changes to the entry in the transaction cache and bypasses the BackingMap value on subsequent operations that are performed on the entry. When a transaction is committed, any entries that are marked as globally invalidated are evicted from the BackingMap. Consider the following use case for invalidation as an example: The BackingMap is backed by a database table that has an auto increment column. Increment columns are useful for assigning unique numbers to records. The application inserts an entry. After the insert, the application needs to know the sequence number for the inserted row. It knows that its copy of the object is old, so it uses global invalidation to get the value from the Loader. The following code demonstrates this use case:

```
Session sess = objectGrid.getSession();
ObjectMap map = sess.getMap("mymap");
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"));
sess.flush();
map.invalidate("Billy", true);
Person p = map.get("Billy");
System.out.println("Version column is: " + p.getVersion());
map.commit();
```

This code sample adds an entry for `Billy`. The version attribute of Person is set using an auto-increment column in the database. The application first performs an insert command. It then issues a flush, which causes the insert to be sent to the Loader and database. The database sets the version column to the next number in the sequence, which makes the Person object in the transaction outdated. To update the object, the application is globally invalidated. The next get method that is issued gets the entry from the Loader, ignoring the transaction value. The entry is fetched from the database with the updated version value.

put method
The semantics of the put method are dependent on whether a previous get method was invoked in the transaction for the key. If the application issues a get operation that returns an entry that exists in the BackingMap or Loader, the put method invocation is interpreted as an update and returns the previous value in the transaction. If a put method invocation ran without a previous get method invocation, or a previous get method invocation did not find an entry, the operation is interpreted as an insert. The semantics of the insert and update methods apply when the put operation is committed. The putAll method is provided to enable batch insert and update processing.

remove method
The remove method removes the entry from the BackingMap and the Loader, if a Loader is plugged in. The value of the object that was removed is returned by this method. If the object does not exist, this method returns a null value. The removeAll method is provided to enable batch deletion processing without the return values.

setCopyMode method
The setCopyMode method specifies a CopyMode value for this ObjectMap. With this method, an application can override the CopyMode value that is specified on the BackingMap. The specified CopyMode value is in effect until clearCopyMode method is invoked. Both methods are invoked outside of transactional bounds. A CopyMode value cannot be changed in the middle of a transaction.

touch method
The touch method updates the last access time for an entry. This method does not retrieve the value from the BackingMap. Use this method in its own transaction. If the provided key does not exist in the BackingMap because of invalidation or removal, an exception occurs during commit processing.

update method

The update method explicitly updates an entry in the BackingMap and the Loader. Using this method indicates to the BackingMap and Loader that you want to update an existing entry. An exception occurs if you invoke this method on an entry that does not exist when the method is invoked or during commit processing.

getIndex method

The getIndex method attempts to obtain a named index that is built on the BackingMap. The index cannot be shared between threads and works on the same rules as a Session. The returned index object should be cast to the right application index interface such as the MapIndex interface, the MapRangeIndex interface, or a custom index interface.

clear method

The clear method removes all cache entries from a map from all partitions. This operation is an auto-commit function, so no active transaction should be present when calling clear.

Note: The clear method only clears out the map on which it is called, leaving any related entity maps unaffected. This method does not invoke the Loader plug-in.

Dynamic maps

With the dynamic maps feature you can create maps after the grid has already been initialized.

In previous versions, eXtreme Scale has required you to define maps before initializing the ObjectGrid. As a result, you had to create all of the maps to be used before running transactions against any of the maps.

Advantages of dynamic maps

The introduction of dynamic maps reduces the restriction of having to define all maps prior to initialization. Through the use of template maps, maps can now be created after the ObjectGrid has been initialized.

Template maps are defined in the ObjectGrid XML file. Template comparisons are run when a Session requests a map that has not been previously defined. If the new map name matches the regular expression of a template map, the map is created dynamically and assigned the name of the requested map. This newly created map inherits all of the settings of the template map as defined by the ObjectGrid XML file.

Creating dynamic maps

Dynamic map creation is tied to the Session.getMap(String) method. Calls to this method return an ObjectMap based on the BackingMap that was configured by the ObjectGrid XML file.

Passing in a String that matches the regular expression of a template map will result in the creation of an ObjectMap and an associated BackingMap.

See the API documentation for more information about the Session.getMap(String cacheName) method.

Defining a template map in XML is as simple as setting a template boolean attribute on the backingMap element. When template is set to true, the name of the backingMap is interpreted as a regular expression.

WebSphere eXtreme Scale uses Java regular expression pattern matching. For more information about the regular expression engine in Java, see the API documentation for the java.util.regex package and classes.

A sample ObjectGrid XML file with a template map defined follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="accounting">
   <backingMap name="payroll" readOnly="false" />
   <backingMap name="templateMap.*" template="true"
    pluginCollectionRef="templatePlugins" lockStrategy="PESSIMISTIC" />
  </objectGrid>
 </objectGrids>

 <backingMapPluginCollections>
  <backingMapPluginCollection id="templatePlugins">
   <bean id="Evictor"
    className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

The previous XML file defines one template map and one non-template map. The name of the template map is a regular expression: `templateMap.*`. When the Session.getMap(String) method is called with a map name matching this regular expression, the application creates a new map.

Note: If you have defined more than one template map, assure that the name of any argument for the Session.getMap(String) method does not match more than one template map.

Example

Configuration of a template map is required in order to create a dynamic map. Add the template boolean to a backingMap in the ObjectGrid XML file.

```
<backingMap name="templateMap.*" template="true" />
```

The name of the template map is treated as a regular expression.

Calling the Session.getMap(String cacheName) method with a cacheName that is a match for the regular expression results in the creation of the dynamic map. An ObjectMap object is returned from this method call, and an associated BackingMap object is created.

```
   Session session = og.getSession();
       ObjectMap map = session.getMap("templateMap1");
```

The newly created map is configured with all the attributes and plug-ins that were defined on the template map definition. Consider again the previous ObjectGrid XML file.

A dynamic map created based on the template map in this XML file would have an evictor configured and its lock strategy would be pessimistic.

Note: A template is not an actual BackingMap. That is, the “accounting” ObjectGrid does not contain an actual “templateMap.*” map. The template is only used as a basis for dynamic map creation. However, you must include the dynamic map in the mapRef element of the deployment policy XML file named exactly as in the ObjectGrid XML. This identifies which mapSet the dynamic maps will be in.

Consider the change in behavior of the Session.getMap(String cacheName) method when using template maps. Before WebSphere eXtreme Scale Version 7.0, all calls to the Session.getMap(String cacheName) method resulted in an UndefinedMapException exception if the map requested did not exist. With dynamic maps, every name that matches the regular expression for a template map results in map creation. Be sure to note the number of maps that your application creates, particularly if your regular expression is generic.

Also, ObjectGridPermission.DYNAMIC_MAP is required for dynamic map creation when eXtreme Scale security is enabled. This permission is checked when the Session.getMap(String) method is called. For more information, see the information about application client authorization in the *Product Overview*.

Limitations and considerations:

Limitations:
- You cannot use dynamic maps with Query.
- The QuerySchema does not support the template for mapName.
- You cannot use entities with dynamic maps.
- An entity BackingMap is implicitly defined, mapped to the entity through the class name.

Considerations:
- Many plug-ins have no way of determining the map with which each plug-in is associated.
- Other plug-ins differentiate themselves by using a map name or BackingMap name as an argument.

ObjectMap and JavaMap

A JavaMap instance is obtained from an ObjectMap object. The JavaMap interface has the same method signatures as ObjectMap, but with different exception handling. JavaMap extends the java.util.Map interface, so all exceptions are instances of the java.lang.RuntimeException class. Because JavaMap extends the java.util.Map interface, it is easy to quickly use WebSphere eXtreme Scale with an existing application that uses a java.util.Map interface for object caching.

Obtain a JavaMap instance

An application gets a JavaMap instance from an ObjectMap object using the ObjectMap.getJavaMap method. The following code snippet demonstrates how to obtain a JavaMap instance.

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
java.util.Map map = objectMap.getJavaMap();
JavaMap javaMap = (JavaMap) javaMap;
```

A JavaMap is backed by the ObjectMap from which it was obtained. Calling the getJavaMap method multiple times using a particular ObjectMap always returns the same JavaMap instance.

Methods

The JavaMap interface only supports a subset of the methods on the java.util.Map interface. The java.util.Map interface supports the following methods:

containsKey(java.lang.Object) method

get(java.lang.Object) method

put(java.lang.Object, java.lang.Object) method

putAll(java.util.Map) method

remove(java.lang.Object) method

clear()

All other methods inherited from the java.util.Map interface result in a java.lang.UnsupportedOperationException exception.

Maps as FIFO queues

With WebSphere eXtreme Scale, you can provide a first-in first-out (FIFO) queue-like capability for all maps. WebSphere eXtreme Scale tracks the insertion order for all maps. A client can ask a map for the next unlocked entry in a map in the order of insertion and lock the entry. This process allows multiple clients to consume entries from the map efficiently.

FIFO example

The following code snippet shows a client entering a loop to process entries from the map until the map is exhausted. The loop starts a transaction, then calls the ObjectMap.getNextKey(5000) method. This method returns the key of the next available unlocked entry and locks it. If the transaction is blocked for more than 5000 milliseconds, then the method returns null.

```
Session session = ...;
ObjectMap map = session.getMap("xxx");
// this needs to be set somewhere to stop this loop
boolean timeToStop = false;

while(!timeToStop)
{
  session.begin();
  Object msgKey = map.getNextKey(5000);
  if(msgKey == null)
  {
    // current partition is exhausted, call it again in
    // a new transaction to move to next partition
    session.rollback();
    continue;
  }
  Message m = (Message)map.get(msgKey);
  // now consume the message
  ...
  // need to remove it
  map.remove(msgKey);
  session.commit();
}
```

Local mode versus client mode

If the application is using a local core, that is, it is not a client, then the mechanism works as described previously.

For client mode, if the Java virtual machine (JVM) is a client, then the client initially connects to a random partition primary. If no work exists in that partition, then the client moves to the next partition to look for work. The client either finds a partition with entries or loops around to the initial random partition. If the client loops around to the initial partition, then it returns a null value to the application. If the client finds a partition with a map that has entries, then it consumes entries from there until no entries are available for the timeout period. After the timeout passes, then null is returned. This action means that when null is returned and a partitioned map is used, then it you should start a new transaction and resume listening. The previous code sample fragment has this behavior.

Example

When you are running as a client and a key is returned, that transaction is now bound to the partition with the entry for that key. If you do not want to update any other maps during that transaction, then a problem does not exist. If you do want to update, then you can only update maps from the same partition as the map from which you got the key. The entry that is returned from the getNextKey method needs to give the application a way to discover relevant data in that partition. As an example, if you have two maps; one for events and another for jobs that the events impact. You define the two maps with the following entities:

Job.java
```
package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;

@Entity
public class Job
{
 @Id String jobId;

 int jobState;
}
```

JobEvent.java
```
package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
import com.ibm.websphere.projector.annotations.OneToOne;

@Entity
public class JobEvent
{
 @Id String eventId;
 @OneToOne Job job;
}
```

The job has as ID and state, which is an integer. Suppose you want to increment the state whenever an event arrived. The events are stored in the JobEvent Map. Each entry has a reference to the job the event concerns. The code for the listener to do this looks like the following example:

JobEventListener.java
```
package tutorial.fifo;

import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class JobEventListener
{
 boolean stopListening;

 public synchronized void stopListening()
 {
  stopListening = true;
 }

 synchronized boolean isStopped()
 {
  return stopListening;
 }

 public void processJobEvents(Session session)
  throws ObjectGridException
 {
  EntityManager em = session.getEntityManager();
  ObjectMap jobEvents = session.getMap("JobEvent");
  while(!isStopped())
  {
   em.getTransaction().begin();

   Object jobEventKey = jobEvents.getNextKey(5000);
   if(jobEventKey == null)
   {
    em.getTransaction().rollback();
    continue;
   }
   JobEvent event = (JobEvent)em.find(JobEvent.class, jobEventKey);
   // process the event, here we just increment the
   // job state
   event.job.jobState++;
   em.getTransaction().commit();
  }
 }
}
```

The listener is started on a thread by the application. The listener runs until the stopListening method is called. The processJobEvents method is run on the thread until the stopListening method is called. The loop blocks waiting for an eventKey from the JobEvent Map and then uses the EntityManager to access the event object, dereference to the job and increment the state.

The EntityManager API does not have a getNextKey method, but the ObjectMap does. So, the code uses the ObjectMap for JobEvent to get the key. If a map is used with entities then it does not store objects anymore. Instead, it stores Tuples; a Tuple object for the key and a Tuple object for the value. The EntityManager.find method accepts a Tuple for the key.

The code to create an event looks like the following example:

```
em.getTransaction().begin();
Job job = em.find(Job.class, "Job Key");
JobEvent event = new JobEvent();
event.id = Random.toString();
event.job = job;
em.persist(event); // insert it
em.getTransaction().commit();
```

You find the job for the event, construct an event, point it to the job, insert it in the JobEvent Map and commit the transaction.

Loaders and FIFO maps

If you want to back a map that is used as a FIFO queue with a Loader, then you might need to do some additional work. If the order of the entries in the map is not a concern, you have no extra work. If the order is important, then you need to add a sequence number to all of the inserted records when you are persisting the records to the backend. The preload mechanism should be written to insert the records on startup using this order.

Caching objects and their relationships (EntityManager API)

Most cache products use map-based APIs to store data as key-value pairs. The ObjectMap API and the dynamic cache in WebSphere Application Server, among others, use this approach. However, map-based APIs have limitations. The EntityManager API simplifies the interaction with the data grid by providing an easy way to declare and interact with a complex graph of related objects.

Map-based API limitations

If you are using a map-based API, such as the dynamic cache in WebSphere Application Server or the ObjectMap API, take the following limitations into consideration:

- Indexes and queries must use reflection to query fields and properties in cache objects.
- Custom data serialization is required to achieve performance for complex objects.
- It is difficult to work with graphs of objects. The application must store artificial references between objects and manually join the objects together.

Benefits of the EntityManager API

The EntityManager API uses the existing map-based infrastructure, but it converts entity objects to and from tuples before storing or reading them from the map. An entity object is transformed into a key tuple and a value tuple, which are then stored as key-value pairs. A tuple is an array of primitive attributes.

This set of APIs follows the Plain Old Java Object (POJO) style of programming that is adopted by most frameworks.

Related reference

API documentation: EntityManager interface

"EntityTransaction interface" on page 82
You can use the EntityTransaction interface to demarcate transactions.

Defining an entity schema

An ObjectGrid can have any number of logical entity schemas. Entities are defined using annotated Java classes, XML, or a combination of both XML and Java classes. Defined entities are then registered with an eXtreme Scale server and bound to BackingMaps, indexes and other plug-ins.

When designing an entity schema, you must complete the following tasks:

1. Define the entities and their relationships.
2. Configure eXtreme Scale.
3. Register the entities.
4. Create entity-based applications that interact with the eXtreme Scale EntityManager APIs.

Entity schema configuration

An entity schema is a set of entities and the relationships between the entities. In an eXtreme Scale application with multiple partitions, the following restrictions and options apply to entity schemas:

- Each entity schema must have a single root defined. This is known as the schema root.
- All the entities for a given schema must be in the same map set, which means that all the entities that are reachable from a schema root with key or non-key relationships must be defined in the same map set as the schema root.
- Each entity can belong to only one entity schema.
- Each eXtreme Scale application can have multiple schemas.

Entities are registered with an ObjectGrid instance before it is initialized. Each defined entity must be uniquely named and is automatically bound to an ObjectGrid BackingMap of the same name. The initialization method varies depending on the configuration you are using:

Local eXtreme Scale configuration

If you are using a local ObjectGrid, you can programmatically configure the entity schema. In this mode, you can use the ObjectGrid.registerEntities methods to register annotated entity classes or an entity metadata descriptor file.

Distributed eXtreme Scale configuration

If you are using a distributed eXtreme Scale configuration, you must provide an entity metadata descriptor file with the entity schema.

For more details, see "EntityManager in a distributed environment" on page 62.

Entity requirements

Entity metadata is configured using Java class files, an entity descriptor XML file or both. At minimum, the entity descriptor XML is required to identify which eXtreme Scale BackingMaps are to be associated with entities. The persistent attributes of the entity and its relationships to other entities are described in either an annotated Java class (entity metadata class) or the entity descriptor XML file. The entity metadata class, when specified, is also used by the EntityManager API to interact with the data in the grid.

7.0.0.0 FIX 2+ An eXtreme Scale grid can be defined without providing any entity classes. This can be beneficial when the server and client are interacting directly with the tuple data stored in the underlying maps. Such entities are defined completely in the entity descriptor XML file and are referred to as classless entities.

Classless entities

Classless entities are useful when it is not possible to include application classes in the server or client classpath. Such entities are defined in the entity metadata descriptor XML file, where the class name of the entity is specified using a classless entity identifier in the form: @<entity identifier>. The @ symbol identifies the entity as classless and is used for mapping associations between entities. See the "Classless entity metadata" figure an example of an entity metadata descriptor XML file with two classless entities defined.

If an eXtreme Scale server or client does not have access to the classes, either can still use the EntityManager API using classless entities. Common use cases include the following:

- The eXtreme Scale container is hosted in a server that does not allow application classes in the classpath. In this case, the clients can still access the grid using the EntityManager API from a client, where the classes are allowed.
- The eXtreme Scale client does not require access to the entity classes because the client is either using a non-Java client, such as the eXtreme Scale REST data service or the client is accessing the tuple data in the grid using the ObjectMap API.

If the entity metadata is compatible between the client and server, entity metadata can be created using entity metadata classes, an XML file, or both.

For example, the "Programmatic entity class" in the following figure is compatible with the classless metadata code in the next section.

Programmatic entity class
```
@Entity
public class Employee {
    @Id long serialNumber;
    @Basic byte[] picture;
    @Version int ver;
    @ManyToOne(fetch=FetchType.EAGER, cascade=CascadeType.PERSIST)
    Department department;
}

@Entity
public static class Department {
    @Id int number;
    @Basic String name;
    @OneToMany(fetch=FetchType.LAZY, cascade=CascadeType.ALL, mappedBy="department")
    Collection<Employee> employees;
}
```

Classless fields, keys, and versions

As previously mentioned, classless entities are configured completely in the entity XML descriptor file. Class-based entities define their attributes using Java fields, properties and annotations. So classless entities need to define key and attribute structure in the entity XML descriptor with the `<basic>` and `<id>` tags.

Classless entity metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

<entity class-name="@Employee" name="Employee">
    <attributes>
        <id name="serialNumber" type="long"/>
        <basic name="firstName" type="java.lang.String"/>
        <basic name="picture" type="[B"/>
        <version name="ver" type="int"/>
        <many-to-one
            name="department"
            target-entity="@Department"
            fetch="EAGER"">
                <cascade><cascade-persist/></cascade>
        </many-to-one>
    </attributes>
</entity>

<entity class-name="@Department" name="Department" >
    <attributes>
        <id name="number" type="int"/>
        <basic name="name" type="java.lang.String"/>
        <version name="ver" type="int"/>
        <one-to-many
            name="employees"
            target-entity="@Employee"
            fetch="LAZY"
            mapped-by="department">
            <cascade><cascade-all/></cascade>
        </one-to-many>
    </attributes>
</entity>
```

Note that each entity above has an <id> element. A classless entity must have either one or more of an <id> element defined, or a single-valued association that represents the key for the entity. The fields of the entity are represented by <basic> elements. The <id>, <version>, and <basic> elements require a name and type in classless entities. See the following supported attribute types section for details on supported types.

Entity class requirements

Class-based entities are identified by associating various metadata with a Java class. The metadata can be specified usingJava Platform, Standard Edition 5 annotations, an entity metadata descriptor file, or a combination of annotations and the descriptor file. Entity classes must meet the following criteria:

- The `@Entity` annotation is specified in the entity XML descriptor file.
- The class has a public or protected no-argument constructor.
- It must be a top-level class. Interfaces and enumerated types are not valid entity classes.
- Cannot use the final keyword.

- Cannot use inheritance.
- Must have a unique name and type for each ObjectGrid instance.

Entities all have a unique name and type. The name, if using annotations, is the simple (short) name of the class by default, but can be overridden using the name attribute of the `@Entity` annotation.

Persistent attributes

The persistent state of an entity is accessed by clients and the entity manager by using either fields (instance variables) or Enterprise JavaBeans-style property accessors. Each entity must define either field- or property-based access. Annotated entities are field-access if the class fields are annotated and are property-access if the getter method of the property is annotated. A mixture of field- and property-access is not allowed. If the type cannot be automatically determined, the **`accessType`** attribute on the `@Entity` annotation or equivalent XML can be used to identify the access type.

Persistent fields

Field-access entity instance variables are accessed directly from the entity manager and clients. Fields that are marked with the transient modifier or transient annotation are ignored. Persistent fields must not have final or static modifiers.

Persistent properties

Property-access entities must adhere to the JavaBeans signature conventions for read and write properties. Methods that do not follow JavaBeans conventions or have the Transient annotation on the getter method are ignored. For a property of type `T`, there must be a getter method getProperty which returns a value of type T and a void setter method setProperty(T). For boolean types, the getter method can be expressed as isProperty, returning true or false. Persistent properties cannot have the static modifier.

Supported attribute types

The following persistent field and property types are supported:

- Java primitive types including wrappers
- java.lang.String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- byte[]
- java.lang.Byte[]
- char[]
- java.lang.Character[]
- enum

User serializable attribute types are supported but have performance, query and change-detection limitations. Persistent data that cannot be proxied, such as arrays and user serializable objects, must be reassigned to the entity if altered.

Serializable attributes are represented in the entity descriptor XML file using the class name of the object. If the object is an array, the data type is represented using the Java internal form. For example, if an attribute data type is java.lang.Byte[][], the string representation is [[Ljava.lang.Byte;

User serializable types should adhere to the following best practices:
- Implement high performance serialization methods. Implement the java.lang.Cloneable interface and public clone method.
- Implement the java.io.Externalizable interface.
- Implement equals and hashCode

Entity associations

Bi-directional and uni-directional entity associations, or relationships between entities can be defined as one-to-one, many-to-one, one-to-many and many-to-many. The entity manager automatically resolves the entity relationships to the appropriate key references when storing the entities.

The eXtreme Scale grid is a data cache and does not enforce referential integrity like a database. Although relationships allow cascading persist and remove operations for child entities, it does not detect or enforce broken links to objects. When removing a child object, the reference to that object must be removed from the parent.

If you define a bi-directional association between two entities, you must identify the owner of the relationship. In a to-many association, the many side of the relationship is always the owning side. If ownership cannot be determined automatically, then the **mappedBy** attribute of the annotation, or XML equivalent, must be specified. The **mappedBy** attribute identifies the field in the target entity that is the owner of the relationship. This attribute also helps identify the related fields when there are multiple attributes of the same type and cardinality.

Single-valued associations

One-to-one and many-to-one associations are denoted using the @OneToOne and @ManyToOne annotations or equivalent XML attributes. The target entity type is determined by the attribute type. The following example defines a uni-directional association between Person and Address. The Customer entity has a reference to one Address entity. In this case, the association could also be many-to-one since there is no inverse relationship.

```
@Entity
public class Customer {
  @Id id;
  @OneToOne Address homeAddress;
}

@Entity
public class Address{
  @Id id
  @Basic String city;
}
```

To specify a bi-directional relationship between the Customer and Address classes, add a reference to the Customer class from the Address class and add the appropriate annotation to mark the inverse side of the relationship. Because this association is one-to-one, you have to specify an owner of the relationship using the mappedBy attribute on the @OneToOne annotation.

```
@Entity
public class Address{
  @Id id
  @Basic String city;
  @OneToOne(mappedBy="homeAddress") Customer customer;
}
```

Collection-valued associations

One-to-many and many-to-many associations are denoted using the `@OneToMany` and `@ManyToMany` annotations or equivalent XML attributes. All many relationships are represented using the types: java.util.Collection, java.util.List or java.util.Set. The target entity type is determined by the generic type of the Collection, List or Set or explicitly using the **`targetEntity`** attribute on the `@OneToMany` or `@ManyToMany` annotation (or XML equivalent).

In the previous example, it is not practical to have one address object per customer because many customers might share an address or might have multiple addresses. This situation is better solved using a many association:

```
@Entity
public class Customer {
  @Id id;
  @ManyToOne Address homeAddress;
  @ManyToOne Address workAddress;
}

@Entity
public class Address{
  @Id id
  @Basic String city;
  @OneToMany(mappedBy="homeAddress") Collection<Customer> homeCustomers;


  @OneToMany(mappedBy="workAddress", targetEntity=Customer.class)
  Collection workCustomers;
}
```

In this example, two different relationships exist between the same entities: a Home and Work address relationship. A non-generic Collection is used for the **`workCustomers`** attribute to demonstrate how to use the **`targetEntity`** attribute when generics are not available.

Classless associations

Classless entity associations are defined in the entity metadata descriptor XML file similar to how class-based associations are defined. The only difference is that instead of the target entity pointing to an actual class, it points to the classless entity identifier used for the class name of the entity.

An example follows:

```
<many-to-one name="department" target-entity="@Department" fetch="EAGER">
      <cascade><cascade-all/></cascade>
</many-to-one>
<one-to-many name="employees" target-entity="@Employee" fetch="LAZY">
      <cascade><cascade-all/></cascade>
</one-to-many>
```

Primary keys

All entities must have a primary key, which can be a simple (single attribute) or composite (multiple attribute) key. The key attributes are denoted using the Id annotation or defined in the entity XML descriptor file. Key attributes have the following requirements:

- The value of a primary key cannot change.
- A primary key attribute should be one of the following types: Java primitive type and wrappers, java.lang.String, java.util.Date or java.sql.Date.
- A primary key can contain any number of single-valued associations. The target entity of the primary key association must not have an inverse association directly or indirectly to the source entity.

Composite primary keys can optionally define a primary key class. An entity is associated with a primary key class using the `@IdClass` annotation or the entity XML descriptor file. An `@IdClass` annotation is useful in conjunction with the EntityManager.find method.

Primary key classes have the following requirements:

- It should be public with a no-argument constructor.
- The access type of the primary key class is determined by the entity that declares the primary key class.
- If property-access, the properties of the primary key class must be public or protected.
- The primary key fields or properties must match the key attribute names and types defined in the referencing entity.
- Primary key classes must implement the equals and hashCode methods.

An example follows:

```
@Entity
@IdClass(CustomerKey.class)
public class Customer {
    @Id @ManyToOne Zone zone;
    @Id int custId;
    String name;
    ...
}

@Entity
public class Zone{
    @Id String zoneCode;
    String name;
}

public class CustomerKey {
    Zone zone;
    int custId;

    public int hashCode() {...}
    public boolean equals(Object o) {...}
}
```

Classless primary keys

Classless entities are required to either have at least one <id> element or an association in the XML file with the attribute `id=true`. An example of both would look like the following:

```
<id name="serialNumber" type="int"/>
<many-to-one name="department" target-entity="@Department" id="true">
<cascade><cascade-all/></cascade>
</many-to-one>
```

Remember:
The <id-class> XML tag is not supported for classless entities.

Entity proxies and field interception

Entity classes and mutable supported attribute types are extended by proxy classes for property-access entities and bytecode-enhanced for Java Development Kit (JDK) 5 field-access entities. All access to the entity, even by internal business methods and the equals methods, must use the appropriate field or property access methods.

Proxies and field interceptors are used to allow the entity manager to track the state of the entity, determine if the entity has changed, and improve performance. Field interceptors are only available on Java SE 5 platforms when the entity instrumentation agent is configured.

Attention: When using property-access entities, the equals method should use the instanceof operator for comparing the current instance to the input object. All introspection of the target object should be through the properties of the object, not the fields themselves, because the object instance will be the proxy.

`emd.xsd` file

Use the entity metadata XML schema definition to create a descriptor XML file and define an entity schema for WebSphere eXtreme Scale.

See the the information about the entity metadata descriptor file in the *Administration Guide* for the descriptions of each element and attribute of the `emd.xsd` file.

`emd.xsd` file

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:emd="http://ibm.com/ws/projector/config/emd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ibm.com/ws/projector/config/emd"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
    version="1.0">

    <!-- **************************************************** -->
    <xsd:element name="entity-mappings">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="description" type="xsd:string" minOccurs="0" />
                <xsd:element name="entity" type="emd:entity" minOccurs="1" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
        <xsd:unique name="uniqueEntityClassName">
            <xsd:selector xpath="emd:entity" />
            <xsd:field xpath="@class-name" />
        </xsd:unique>
    </xsd:element>

    <!-- **************************************************** -->
    <xsd:complexType name="entity">
        <xsd:sequence>
            <xsd:element name="description" type="xsd:string" minOccurs="0" />
            <xsd:element name="id-class" type="emd:id-class" minOccurs="0" />
            <xsd:element name="attributes" type="emd:attributes" minOccurs="0" />
            <xsd:element name="entity-listeners" type="emd:entity-listeners" minOccurs="0" />
            <xsd:element name="pre-persist" type="emd:pre-persist" minOccurs="0" />
            <xsd:element name="post-persist" type="emd:post-persist" minOccurs="0" />
            <xsd:element name="pre-remove" type="emd:pre-remove" minOccurs="0" />
            <xsd:element name="post-remove" type="emd:post-remove" minOccurs="0" />
            <xsd:element name="pre-invalidate" type="emd:pre-invalidate" minOccurs="0" />
            <xsd:element name="post-invalidate" type="emd:post-invalidate" minOccurs="0" />
            <xsd:element name="pre-update" type="emd:pre-update" minOccurs="0" />
            <xsd:element name="post-update" type="emd:post-update" minOccurs="0" />
            <xsd:element name="post-load" type="emd:post-load" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="class-name" type="xsd:string" use="required" />
        <xsd:attribute name="access" type="emd:access-type" />
        <xsd:attribute name="schemaRoot" type="xsd:boolean" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="attributes">
        <xsd:sequence>
            <xsd:choice>
                <xsd:element name="id" type="emd:id" minOccurs="0" maxOccurs="unbounded" />
            </xsd:choice>
            <xsd:element name="basic" type="emd:basic" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="version" type="emd:version" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="many-to-one" type="emd:many-to-one" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="one-to-many" type="emd:one-to-many" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="one-to-one" type="emd:one-to-one" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="many-to-many" type="emd:many-to-many" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="transient" type="emd:transient" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:simpleType name="access-type">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="PROPERTY" />
            <xsd:enumeration value="FIELD" />
        </xsd:restriction>
    </xsd:simpleType>

    <!-- **************************************************** -->
    <xsd:complexType name="id-class">
        <xsd:attribute name="class-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="id">
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="type" type="xsd:string" />
        <xsd:attribute name="alias" type="xsd:string" use="optional" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="transient">
        <xsd:attribute name="name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="basic">
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="alias" type="xsd:string" />
        <xsd:attribute name="type" type="xsd:string" />
        <xsd:attribute name="fetch" type="emd:fetch-type" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:simpleType name="fetch-type">
        <xsd:restriction base="xsd:token">
            <xsd:enumeration value="LAZY" />
            <xsd:enumeration value="EAGER" />
        </xsd:restriction>
    </xsd:simpleType>

    <!-- **************************************************** -->
    <xsd:complexType name="many-to-one">
        <xsd:sequence>
            <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="alias" type="xsd:string" />
        <xsd:attribute name="target-entity" type="xsd:string" />
        <xsd:attribute name="fetch" type="emd:fetch-type" />
        <xsd:attribute name="id" type="xsd:boolean" />
    </xsd:complexType>
    <!-- **************************************************** -->
    <xsd:complexType name="one-to-one">
        <xsd:sequence>
            <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="alias" type="xsd:string" />
        <xsd:attribute name="target-entity" type="xsd:string" />
        <xsd:attribute name="fetch" type="emd:fetch-type" />
        <xsd:attribute name="mapped-by" type="xsd:string" />
        <xsd:attribute name="id" type="xsd:boolean" />
    </xsd:complexType>
    <!-- **************************************************** -->
    <xsd:complexType name="one-to-many">
        <xsd:sequence>
            <xsd:element name="order-by" type="emd:order-by" minOccurs="0" />
            <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="alias" type="xsd:string" />
        <xsd:attribute name="target-entity" type="xsd:string" />
        <xsd:attribute name="fetch" type="emd:fetch-type" />
        <xsd:attribute name="mapped-by" type="xsd:string" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="many-to-many">
        <xsd:sequence>
            <xsd:element name="order-by" type="emd:order-by" minOccurs="0" />
            <xsd:element name="cascade" type="emd:cascade-type" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required" />
        <xsd:attribute name="alias" type="xsd:string" />
        <xsd:attribute name="target-entity" type="xsd:string" />
        <xsd:attribute name="fetch" type="emd:fetch-type" />
        <xsd:attribute name="mapped-by" type="xsd:string" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:simpleType name="order-by">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>

    <!-- **************************************************** -->
    <xsd:complexType name="cascade-type">
        <xsd:sequence>
            <xsd:element name="cascade-all" type="emd:emptyType" minOccurs="0" />
            <xsd:element name="cascade-persist" type="emd:emptyType" minOccurs="0" />
            <xsd:element name="cascade-remove" type="emd:emptyType" minOccurs="0" />
            <xsd:element name="cascade-invalidate" type="emd:emptyType" minOccurs="0" />
            <xsd:element name="cascade-merge" type="emd:emptyType" minOccurs="0" />
            <xsd:element name="cascade-refresh" type="emd:emptyType" minOccurs="0" />
        </xsd:sequence>
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="emptyType" />

    <!-- **************************************************** -->
    <xsd:complexType name="version">
          <xsd:attribute name="name" type="xsd:string" use="required"/>
          <xsd:attribute name="alias" type="xsd:string" />
          <xsd:attribute name="type" type="xsd:string" />
    </xsd:complexType>

    <!-- **************************************************** -->

    <xsd:complexType name="entity-listeners">
        <xsd:sequence>
            <xsd:element name="entity-listener" type="emd:entity-listener" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="entity-listener">
        <xsd:sequence>
            <xsd:element name="pre-persist" type="emd:pre-persist" minOccurs="0" />
            <xsd:element name="post-persist" type="emd:post-persist" minOccurs="0" />
            <xsd:element name="pre-remove" type="emd:pre-remove" minOccurs="0" />
            <xsd:element name="post-remove" type="emd:post-remove" minOccurs="0" />
            <xsd:element name="pre-invalidate" type="emd:pre-invalidate" minOccurs="0" />
            <xsd:element name="post-invalidate" type="emd:post-invalidate" minOccurs="0" />
            <xsd:element name="pre-update" type="emd:pre-update" minOccurs="0" />
            <xsd:element name="post-update" type="emd:post-update" minOccurs="0" />
            <xsd:element name="post-load" type="emd:post-load" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="class-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="pre-persist">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- **************************************************** -->
    <xsd:complexType name="post-persist">
```

```
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="pre-remove">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="post-remove">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="pre-invalidate">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="post-invalidate">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="pre-update">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="post-update">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

    <!-- ****************************************************** -->
    <xsd:complexType name="post-load">
        <xsd:attribute name="method-name" type="xsd:string" use="required" />
    </xsd:complexType>

</xsd:schema>
```

EntityManager in a distributed environment

You can use EntityManager with a local ObjectGrid or in a distributed eXtreme Scale environment. The main difference is how you connect to this remote environment. After you establish a connection, there is no difference between using a Session object or using the EntityManager API.

Required configuration files

The following XML configuration files are required:
- ObjectGrid descriptor XML file
- Entity descriptor XML file
- Deployment or data grid descriptor XML file

These files specify which entities and BackingMaps a server will host.

The entity metadata descriptor file contains a description of the entities that are used. At minimum, you must specify the entity class and name. If you are running in a Java Platform, Standard Edition 5 environment, eXtreme Scale automatically reads the entity class and its annotations. You can define additional XML attributes if the entity class has no annotations or if you are required to override the class attributes. If you are registering the entities classless , provide all of entity information in the XML file only.

You can use the following XML configuration snippet to define an eXtreme Scale data grid with entities. In this snippet, the server creates an ObjectGrid with the name `bookstore` and an associated backing map with the name `order`. Note that the objectgrid.xml snippet refers to the `entity.xml` file. In this case, the `entity.xml` file contains one entity, the Order entity.

objectgrid.xml
```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">

 <objectGrids>
    <objectGrid name="bookstore" entityMetadataXMLFile="entity.xml">
     <backingMap name="Order"/>
    </objectGrid>
</objectGrids>

</objectGridConfig>
```

This `objectgrid.xml` file refers to the `entity.xml` with the entityMetadataXMLFile attribute. The location of this file is relative to the location of the `objectgrid.xml` file. An example of the `entity.xml` file follows:

entity.xml
```
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">
 <entity class-name="com.ibm.websphere.tutorials.objectgrid.em.
    distributed.step1.Order" name="Order"/>
</entity-mappings>
```

This example assumes the Order class would have the orderNumber and desc fields annotated similarly.

An equivalent classless entity.xml would be as follows

classless entity.xml
```
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">
 <entity class-name="@Order " name="Order">
     <description>"Entity named: Order"</description>
        <attributes>
            <id name="orderNumber" type="int"/>
            <basic name="desc" type="java.lang.String"/>
        </attributes>
 </entity>
</entity-mappings>
```

For information about starting an eXtreme Scale server, see *Starting WebSphere eXtreme Scale server processes* in the *Administration Guide*, which uses both the `deployment.xml` and `objectgrid.xml` files to start the catalog server.

Connecting to a distributed eXtreme Scale server

The following code enables the connect mechanism for a client and server on the same computer:

```
String catalogEndpoints="localhost:2809";
URL clientOverrideURL= new URL("file:etc/emtutorial/distributed/step1/objectgrid.xml");
ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, clientOverrideURL);
ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
```

In the preceding code snippet, note the reference to the remote eXtreme Scale server. After you establish a connection , you can invoke EntityManager API methods such as persist, update, remove and find.

Attention: When you are using entities, pass the client override ObjectGrid descriptor XML file to the connect method. If a null value is passed to the clientOverrideURL property and the client has a different directory structure than the server, then the client might fail to locate the ObjectGrid or entity descriptor XML files. At minimum, the ObjectGrid and entity XML files for the server can be copied to the client.

Previously, using entities on an ObjectGrid client required you to make the ObjectGrid XML and entity XML available to the client in one of the following two ways:

1. Pass an overriding ObjectGrid XML to the ObjectGridManager.connect(String catalogServerAddresses, ClientSecurityConfiguration securityProps, URL overRideObjectGridXml) method.

   ```
   String catalogEndpoints="myHost:2809";
   URL clientOverrideURL= new URL("file:etc/emtutorial/distributed/step1/objectgrid.xml");
   ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, clientOverrideURL);
   ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
   ```

2. Pass null for the override file and ensure that the ObjectGrid XML and referenced entity XML are available to the client on the same path as on the server.

   ```
   String catalogEndpoints="myHost:2809";
   ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, null);
   ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
   ```

7.0.0.0 FIX 2+ The XML files were required regardless of whether or not you wanted to use subset entities on the client side. These files are no longer required to use the entities as defined by the server. Instead, pass null as the overRideObjectGridXml parameter as in option 2 of the previous section. If the XML file is not found on the same path set on the server, the client will use the entity configuration on the server.

However, if you use subset entities on the client, you must provide an overriding ObjectGrid XML as in option 1.

Client and server side schema

The server-side schema defines the type of data stored in the maps on a server. The client-side schema is a mapping to application objects from the schema on the server. For example, you might have the following server-side schema:

```
@Entity
class ServerPerson
{
  @Id String ssn;
  String firstName;
  String surname;
  int age;
  int salary;
}
```

A client can have an object annotated as in the following example:

```
@Entity(name="ServerPerson")
class ClientPerson
{
  @Id @Basic(alias="ssn") String socialSecurityNumber;
  String surname;
}
```

This client then takes a server-side entity and projects the subset of the entity into the client object. This projection leads to bandwidth and memory savings on a client because the client has only the information it needs instead of all of the information that is in the server side entity. Different applications can use their own objects instead of forcing all applications to share a set of classes for data access.

The client-side entity descriptor XML file is required in the following cases: if the server is running with class-based entities while the client side is running classless; or if the server is classless and the client uses class-based entities. A classless client mode allows the client to still run entity queries without having access to the physical classes. Assuming the server has registered the ServerPerson entity above, the client would override the data grid with an entity.xml such as follows:

```
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">
 <entity class-name="@ServerPerson " name="Order">
     <description>"Entity named: Order"</description>
        <attributes>
            <id name="socialSecurityNumber" type="java.lang.String"/>
            <basic name="surname" type="java.lang.String"/>
        </attributes>
 </entity>
</entity-mappings>
```

This will achieve an equivalent subset entity on the client, without requiring the client to provide the actual annotated class. If the server is classless, and the client is not, the client overrides with an entity descriptor XML file with a reference to the class file.

Referencing the schema

If running in Java SE 5, then the application can be added to the objects by using annotations. The EntityManager can read the schema from the annotations on those objects. The application provides the eXtreme Scale run time with references to these objects using the `entity.xml` file, which is referenced from the `objectgrid.xml` file. The `entity.xml` file lists all the entities, each of which is associated with either a class or a schema. If a proper class name is specified, then the application attempts to read the Java SE 5 annotations from those classes to determine the schema. If you do not annotate the class file or specify a classless identifier as the class name, then the schema is taken from the XML file. The XML file is used to specify all the attributes, keys, and relationships for each entity.

A local data grid does not need XML files. The program can obtain an ObjectGrid reference and invoke the ObjectGrid.registerEntities method to specify a list of Java SE 5 annotated classes or an XML file.

The run time uses the XML file or a list of annotated classes to find entity names, attribute names and types, key fields and types, and relationships between entities. If eXtreme Scale is running on a server or in stand-alone mode, then it automatically makes a map named after each entity. These maps can be customized further using the `objectgrid.xml` file or APIs set either by the application or injection frameworks such as Spring.

Entity metadata descriptor file

See “`emd.xsd` file” on page 59 for more information about the metadata descriptor file.

Related reference

API documentation: EntityManager interface

“EntityTransaction interface” on page 82
You can use the EntityTransaction interface to demarcate transactions.

Interacting with EntityManager

Applications typically first obtain an ObjectGrid reference, and then a Session from that reference for each thread. Sessions cannot be shared between threads. An extra method on Session, the getEntityManager method, is available. This method returns a reference to an entity manager to use for this thread. The EntityManager interface can replace the Session and ObjectMap interfaces for all applications. You can use these EntityManager APIs if the client has access to the defined entity classes.

Obtaining an EntityManager instance from a session

The getEntityManager method is available on a Session object. The following code example illustrates how to create a local ObjectGrid instance and access the EntityManager. See the EntityManager interface in the API documentation for details about all the supported methods.

```
ObjectGrid og =
ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("intro-grid");
Session s = og.getSession();
EntityManager em = s.getEntityManager();
```

A one-to-one relationship exists between the Session object and EntityManager object. You can use the EntityManager object more than once.

Persisting an entity

Persisting an entity means saving the state of a new entity in an ObjectGrid cache. After the persist method is called, the entity is in the managed state. Persist is a transactional operation, and the new entity is stored in the ObjectGrid cache after the transaction commits.

Every entity has a corresponding BackingMap in which the tuples are stored. The BackingMap has the same name as the entity, and is created when the class is registered. The following code example demonstrates how to create an Order object by using the persist operation.

```
Order order = new Order(123);
em.persist(order);
order.setX();
...
```

The Order object is created with the key 123, and the object is passed to the persist method. You can continue to modify the state of the object before you commit the transaction.

Important: The preceding example does not include any required transactional boundaries, such as begin and commit. See the the entity manager tutorial in the *Product Overview* for more information.

Finding an entity

You can locate the entity in the ObjectGrid cache with the find method by providing a key after the entity is stored in the cache. This method does not require any transactional boundary, which is useful for read-only semantics. The following example illustrates that only one line of code is needed to locate the entity.

```
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
```

Removing an entity

The remove method, like the persist method, is a transactional operation. The following example shows the transactional boundary by calling the begin and commit methods.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.remove(foundOrder );
em.getTransaction().commit();
```

The entity must first be managed before it can be removed, which you can accomplish by calling the find method within the transactional boundary. Then call the remove method on the EntityManager interface.

Invalidating an entity

The invalidate method behaves much like the remove method, but does not invoke any Loader plug-ins. Use this method to remove entities from the ObjectGrid, but to preserve them in the backend data store.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.invalidate(foundOrder );
em.getTransaction().commit();
```

The entity must first be managed before it can be invalidated, which you can accomplish by calling the find method within the transactional boundary. After you call the find method, you can call the invalidate method on the EntityManager interface.

Updating an entity

The update method is also a transactional operation. The entity must be managed before any updates can be applied.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
foundOrder.date = new Date(); // update the date of the order
em.getTransaction().commit();
```

In the preceding example, the persist method is not called after the entity is updated. The entity is updated in the ObjectGrid cache when the transaction is committed.

Queries and query queues

With the flexible query engine, you can retrieve entities by using EntityManager API. Create SELECT type queries over an entity or Object-based schema by using the ObjectGrid query language. Query interface explains in detail how you can run the queries by using the EntityManager API. See the Query API for more information about using queries.

An entity QueryQueue is a queue-like data structure associated with an entity query. It selects all the entities that match the WHERE condition on the query filter and puts the result entities in a queue. Clients can then iteratively retrieve entities from this queue. See “Entity query queues” on page 79for more information.

Entity listeners and callback methods

Applications can be notified when the state of an entity transitions from state to state. Two callback mechanisms exist for state change events: life cycle callback methods that are defined on an entity class and are invoked whenever the entity state changes, and entity listeners, which are more general because the entity listener can be registered on several entities.

Life cycle of an entity instance

An entity instance has the following states:

- **New**: A newly created entity instance that does not exist in the eXtreme Scale cache.
- **Managed**: The entity instance exists in the eXtreme Scale cache and is retrieved or persisted using the entity manager. An entity must be associated with an active transaction to be in the managed state.
- **Detached**: The entity instance exists in the eXtreme Scale cache, but is no longer associated with an active transaction.
- **Removed**: The entity instance is removed, or is scheduled to be removed, from the eXtreme Scale cache when the transaction is flushed or committed.
- **Invalidated**: The entity instance is invalidated, or is scheduled to be invalidated, from the eXtreme Scale cache when the transaction is flushed or committed.

When entities change from state to state, you can invoke life-cycle, call-back methods.

The following sections further describe the meanings of New, Managed, Detached, Removed and Invalidated states as the states apply to an entity.

Entity life cycle callback methods

Entity life cycle callback methods can be defined on the entity class and are invoked when the entity state changes. These methods are useful for validating entity fields and updating transient state that is not usually persisted with the entity. Entity life cycle callback methods can also be defined on classes that are not using entities. Such classes are entity listener classes, which can be associated with multiple entity types. life cycle callback methods can be defined using both metadata annotations and a entity metadata XML descriptor file:

- **Annotations**: life cycle callback methods can be denoted using the PrePersist, PostPersist, PreRemove, PostRemove, PreUpdate, PostUpdate, and PostLoad annotations in an entity class.
- **Entity XML descriptor** : life cycle callback methods can be described using XML when annotations are not available.

Entity listeners

An entity listener class is a class that does not use entities that defines one or more entity life cycle callback methods. Entity listeners are useful for general purpose auditing or logging applications. Entity listeners can be defined using both metadata annotations and a entity metadata XML descriptor file:

- **Annotation**: The EntityListeners annotation can be used to denote one or more entity listener classes on an entity class. If multiple entity listeners are defined, the order in which they are invoked is determined by the order in which they are specified in the EntityListeners annotation.
- **Entity XML descriptor**: The XML descriptor can be used as an alternative to specify the invocation order of entity listeners or to override the order that is specified in metadata annotations.

Callback method requirements

Any subset or combination of annotations can be specified on an entity class or a listener class. A single class cannot have more than one life cycle callback method for the same life cycle event. However, the same method can be used for multiple callback events. The entity listener class must have a public no-arg constructor. Entity listeners are stateless. The life cycle of an entity listener is unspecified. eXtreme Scale does not support entity inheritance, so callback methods can only be defined in the entity class, but not in the superclass.

Callback method signature

Entity life cycle callback methods can be defined on an entity listener class, directly on an entity class, or both. Entity life cycle callback methods can be defined using both metadata annotations and the entity XML descriptor. The annotations used for callback methods on the entity class and on the entity listener class are the same. The signatures of the callback methods are different when defined on an entity class versus an entity listener class. Callback methods defined on an entity class or mapped superclass have the following signature:

```
void <METHOD>()
```

Callback methods that are defined on an entity listener class have the following signature:

```
void <METHOD>(Object)
```

The Object argument is the entity instance for which the callback method is invoked. The Object argument can be declared as a java.lang.Object object or the actual entity type.

Callback methods can have public, private, protected, or package level access, but must not be static or final.

The following annotations are defined to designate life cycle event callback methods of the corresponding types:

- com.ibm.websphere.projector.annotations.PrePersist
- com.ibm.websphere.projector.annotations.PostPersist
- com.ibm.websphere.projector.annotations.PreRemove
- com.ibm.websphere.projector.annotations.PostRemove
- com.ibm.websphere.projector.annotations.PreUpdate
- com.ibm.websphere.projector.annotations.PostUpdate
- com.ibm.websphere.projector.annotations.PostLoad

See the API Documentation for more details. Each annotation has an equivalent XML attribute defined in the entity metadata XML descriptor file.

Life cycle callback method semantics

Each of the different life cycle callback methods has a different purpose and is called in different phases of the entity life cycle:

PrePersist
: Invoked for an entity before the entity has been persisted to the store, which includes entities that have been persisted due to a cascading operation. This method is invoked on the thread of the EntityManager.persist operation.

PostPersist
: Invoked for an entity after the entity has been persisted to the store, which includes entities that have been persisted due to a cascading operation.

This method is invoked on the thread of the EntityManager.persist operation. It is called after the EntityManager.flush or EntityManager.commit is called.

PreRemove
Invoked for an entity before the entity has been removed, which includes entities that have been removed due to a cascading operation. This method is invoked on the thread of the EntityManager.remove operation.

PostRemove
Invoked for an entity after the entity has been removed, which includes entities that have been removed due to a cascading operation. This method is invoked on the thread of the EntityManager.remove operation. It is called after the EntityManager.flush or EntityManager.commit is called.

PreUpdate
Invoked for an entity before the entity has been updated to the store. This method is invoked on the thread of the transaction flush or commit operation.

PostUpdate
Invoked for an entity after the entity has been updated to the store. This method is invoked on the thread of the transaction flush or commit operation.

PostLoad
Invoked for an entity after the entity has been loaded from the store which includes any entities that are loaded through an association. This method is invoked on the thread of the loading operation, such as EntityManager.find or a query.

Duplicate life cycle callback methods

If multiple callback methods are defined for an entity life cycle event, the ordering of the invocation of these methods is as follows:

1. **life cycle callback methods defined in the entity listeners**: The life cycle callback methods that are defined on the entity listener classes for an entity class are invoked in the same order as the specification of the entity listener classes in the EntityListeners annotation or the XML descriptor.
2. **Listener super class**: Callback methods defined in the super class of the entity listener are invoked before the children.
3. **Entity life cycle methods**: WebSphere eXtreme Scale does not support entity inheritance, so the entity life cycle methods can only be defined in the entity class.

Exceptions

Life cycle callback methods might result in run time exceptions. If a life cycle callback method results in a run time exception within a transaction, the transaction is rolled back. No further life cycle callback methods are invoked after a runtime exception results.

Entity listener examples

You can write EntityListeners based on your requirements. Several example scripts follow.

EntityListeners example using annotations

The following example shows the life-cycle callback method invocations and order of the invocations. Assume an entity class Employee and two entity listeners exist: EmployeeListener and EmployeeListener2.

```
@Entity
@EntityListeners(EmployeeListener.class, EmployeeListener2.class)
public class Employee {
    @PrePersist
    public void checkEmployeeID() {
        ....
    }
}

public class EmployeeListener {
    @PrePersist
    public void onEmployeePrePersist(Employee e) {
        ....
    }
}

public class PersonListener {
    @PrePersist
    public void onPersonPrePersist(Object person) {
        ....
    }
}

public class EmployeeListener2 {
    @PrePersist
    public void onEmployeePrePersist2(Object employee) {
        ....
    }
}
```

If a PrePersist event occurs on an Employee instance, the following methods are called in order:

1. onEmployeePrePersist method
2. onPersonPrePersist method
3. onEmployeePrePersist2 method
4. checkEmployeeID method

Entity listeners example using XML

The following example shows how to set an entity listener on an entity using the entity descriptor XML file:

```
<entity
    class-name="com.ibm.websphere.objectgrid.sample.Employee"
    name="Employee" access="FIELD">
    <attributes>
        <id name="id" />
        <basic name="value" />
    </attributes>
    <entity-listeners>
        <entity-listener
            class-name="com.ibm.websphere.objectgrid.sample.EmployeeListener">
            <pre-persist method-name="onListenerPrePersist" />
            <post-persist method-name="onListenerPostPersist" />
        </entity-listener>
    </entity-listeners>
    <pre-persist method-name="checkEmployeeID" />
</entity>
```

The entity Employee is configured with a com.ibm.websphere.objectgrid.sample.EmployeeListener entity listener class , which has two life-cycle callback methods defined. The onListenerPrePersist method is for the PrePersist event, and the onListenerPostPersist method is for the PostPersist event. Also, the checkEmployeeID method in the Employee class is configured to listen for the PrePersist event.

EntityManager fetch plan support

A FetchPlan is the strategy the eXtreme Scale EntityManager uses for retrieving associated objects if the application needs to access relationships.

Example

Assume for example that your application has two entities: Department and Employee. The relationship between the Department entity and the Employee entity is a bi-directional one-to-many relationship: One department has many employees, and one employee belongs to only one department. Since most of the time, when Department entity is fetched, its employees are likely to be fetched, the fetch type of this one-to-many relationship is set to be EAGER.

Here is a snippet of the Department class.

```
 @Entity
public class Department {

    @Id
    private String deptId;

    @Basic
    String deptName;

  @OneToMany(fetch = FetchType.EAGER, mappedBy="department", cascade = {CascadeType.PERSIST})
  public Collection<Employee> employees;

}
```

In a distributed environment, when an application calls `em.find(Department.class, "dept1")` to find a Department entity with key "dept1", this find operation will get the Department entity and all its eager-fetched relations. In the case of the preceding snippet, these are all the employees of department "dept1".

Prior to WebSphere eXtreme Scale 6.1.0.5, the retrieval of one Department entity and N Employee entities incurred N+1 client-server trips because the client retrieved one entity for one client-server trip. You can improve performance if you retrieve these N+1 entities in one trip.

Fetch plan

A fetch plan can be used to customize how to fetch eager relationships by customizing the maximum depth of the relationships. The fetch depth overrides eager relations greater than the specified depth to lazy relations. By default, the fetch depth is the maximum fetch depth. This means that eager relationships of all levels that are eager-navigable from the root entity will be fetched. An EAGER relationship is eager-navigable from a root entity if and only if all the relations starting from the root entity to it are configured as eager-fetched.

In the previous example, the Employee entity is eager-navigable from the Department entity because the Department-Employee relationship is configured as eager-fetched.

If the Employee entity has another eager relationship to an Address entity for instance, then the Address entity is also eager-navigable from the Department entity. However, if the Department-Employee relationships were configured as lazy-fetch, then the Address entity is not eager-navigable from the Department entity because the Department-Employee relationship breaks the eager fetch chain.

A FetchPlan object can be retrieved from the EntityManager instance. The application can use the setMaxFetchDepth method to change the maximum fetch depth.

A fetch plan is associated with an EntityManager instance. The fetch plan applies to any fetch operation, more specifically as follows.

- EntityManager `find(Class class, Object key)` and `findForUpdate(Class class, Object key)` operations
- Query operations
- QueryQueue operations

The FetchPlan object is mutable. Once changed, the changed value will be applied to the fetch operations executed afterward.

A fetch plan is important for a distributed deployment because it decides whether the eager-fetched relationship entities are retrieved with the root entity in one client-server trip or more than one.

Continuing with the previous example, consider further that the fetch plan has maximum depth set to infinity. In that case, when an application calls `em.find(Department.class, "dept1")` to find a Department, this find operation will get one Department entity and N employee entities in one client-server trip. However, for a fetch plan with maximum fetch depth set to zero, only the Department object will be retrieved from the server, while the Employee entities are retrieved from the server only when the employees collection of the Department object is accessed.

Different fetch plans

You have several different fetch plans based on your requirements, explained in the following sections.

Impact on a distributed grid

- *Infinite-depth fetch plan:* An infinite-depth fetch plan has its maximum fetch depth set to `FetchPlan.DEPTH_INFINITE`.

  In a client-server environment, if an infinite-depth fetch plan is used, then all the relations that are eager-navigable from the root entity will be retrieved in one client-server trip.

  Example: If the application is interested in all the Address entities of all employees of a particular Department, then it uses an infinite-depth fetch plan to retrieve all the associated Address entities. The following code only incurs one client-server trip.

```
em.getFetchPlan().setMaxFetchDepth(FetchPlan.DEPTH_INFINITE);

tran.begin();
Department dept = (Department) em.find(Department.class, "dept1");
// do something with Address object.
for (Employee e: dept.employees) {
    for (Address addr: e.addresses) {
```

```
    // do something with addresses.
    }
}
tran.commit();
```

- *Zero-depth fetch plan:* A zero-depth fetch plan has its maximum fetch depth set to 0.

  In a client-server environment, if a zero fetch plan is used, then only the root entity will be retrieved in the first client-server trip. All the eager relationships are treated as if they were lazy.

  Example: In this example, the application is only interested in the Department entity attribute. It does not need to access its employees, so the application sets the fetch plan depth to 0.

  ```
  Session session = objectGrid.getSession();
  EntityManager em = session.getEntityManager();
  EntityTransaction tran = em.getTransaction();
  em.getFetchPlan().setMaxFetchDepth(0);

  tran.begin();
  Department dept = (Department) em.find(Department.class, "dept1");
  // do something with dept object.
  tran.commit();
  ```

- *Fetch plan with depth* k:

  A *k*-depth fetch plan has its maximum fetch depth set to k.

  In a client-server eXtreme Scale environment, if a *k*-depth fetch plan is used, then all the relationships eager-navigable from the root entity within *k* steps will be retrieved in the first client-server trip.

  The infinite-depth fetch plan (*k* = infinity) and zero-depth fetch plan (*k* = 0) are just two examples of the *k*-depth fetch plan.

  To continue expanding on the previous example, assume there is another eager relationship from the entity Employee to the entity Address. If the fetch plan has maximum fetch depth set to 1, then the `em.find(Department.class, "dept1")` operation will retrieve the Department entity and all its Employee entities in one client-server trip. However, the Address entities will not be retrieved because they are not eager-navigable to the Department entity within 1 step, but 2 steps.

  If you use a fetch plan with depth set to 2, then the `em.find(Department.class, "dept1")` operation will retrieve the Department entity, all its Employee entities, and all Address entities associated with the Employee entities in one client-server trip.

  Tip: The default fetch plan has maximum fetch depth set to infinity, so the default behavior of a fetch operation can change. All the eager-navigable relationships from the root entity are retrieved. Instead of multiple trips, now the fetch operation only incurs one client-server trip with the default fetch plan. To keep the settings for the product from the prior version, set the fetch depth to 0.

- *Fetch plan used on query:*

  If you execute an entity query you can also use a fetch plan to customize relationship retrieval.

  For example, the query `SELECT d FROM Department d WHERE "d.deptName='Department'"` result has a relationship to the Department entity. Notice the fetch plan depth starts with the query result association: In this case, the Department entity, not the query result itself. That is, the Department entity is on fetch-depth level 0. Therefore a fetch plan with maximum fetch depth 1 will retrieve the Department entity and its Employee entities in one client-server trip.

Example: In this example, the fetch plan depth is set to 1, so the Department entity and its Employee entities are retrieved in one client-server trip, but the Address entities will not be retrieved in the same trip.

Important: If a relationship is ordered, using either OrderBy annotation or configuration, then it is considered an eager relationship even if it is configured as lazy-fetch.

Performance considerations in a distributed environment

By default, all relationships that are eager-navigable from the root entity will be retrieved in one client-server trip. This can improve performance if all the relationships are going to be used. However, in certain usage scenarios, not all relationships eager-navigable from the root entity are used, so they incur both run-time overhead and bandwidth overhead by retrieving those unused entities.

For such cases, the application can set the maximum fetch depth to a small number to decrease the depth of entities to be retrieved by making all the eager relations after that certain depth lazy. This setting can improve performance.

Proceeding still further with the previous Department-Employee-Address example, by default, all the Address entities associated with employees of the Department "dept1" will be retrieved when `em.find(Department.class, "dept1")` is called. If the application does not use Address entities, it can set the maximum fetch depth to 1, so the Address entities will not be retrieved with the Department entity.

EntityManager interface performance impact

An environment requiring every application to use the same data access objects for a given datastore would be highly impractical. In contrast, the EntityManager interface that is provided with WebSphere eXtreme Scale separates applications from the state held in its server grid datastore.

The cost of using the EntityManager interface is not high and depends on the type of work being performed. Always use the EntityManager interface and optimize the crucial business logic after the application is complete. You can rework any code that uses EntityManager interfaces to use maps and tuples. Generally, this code rework might be necessary for ten percent of the code.

If you use relationships between objects, then the performance impact is lower because an application that is using maps needs to manage those relationships similarly to the EntityManager interface.

Applications that use the EntityManager interface do not need to provide an ObjectTransformer because it is optimized automatically.

Reworking EntityManager code for maps

A sample entity follows:

```
@Entity
public class Person
{
 @Id
 String ssn;
 String firstName;
 @Index
 String middleName;
 String surname;
}
```

Some code to find the entity and update the entity follows:

```
Person p = null;
s.begin();
p = (Person)em.find(Person.class, "1234567890");
p.middleName = String.valueOf(inner);
s.commit();
```

The same code using Maps and Tuples follows:

```
Tuple key = null;
key = map.getEntityMetadata().getKeyMetadata().createTuple();
key.setAttribute(0, "1234567890");

// The Copy Mode is always NO_COPY for entity maps if not using COPY_TO_BYTES.
// Either we need to copy the tuple or we can ask the ObjectGrid to do it for us:
map.setCopyMode(CopyMode.COPY_ON_READ);
s.begin();
Tuple value = (Tuple)map.get(key);
value.setAttribute(1, String.valueOf(inner));
map.update(key, value);
value = null;
s.commit();
```

Both of these code snippets have the same result, and an application can use either or both snippets.

The second code snippet shows how to use maps directly and how to work with the tuples (the key and value pairs). The value tuple has three attributes: firstName, middlename, and surname, indexed at 0, 1, and 2 respectively. The key tuple has a single attribute the ID number is indexed at zero. You can see how Tuples are created by using the EntityMetadata#getKeyMetaData or EntityMetadata#getValueMetaData methods. You must use these methods to create Tuples for an Entity. You cannot implement the Tuple interface and pass an instance of your Tuple implementation.

Instrumentation agent

You can improve the performance of field-access entities by enabling the WebSphere eXtreme Scale instrumentation agent when using Java Development Kit (JDK) Version 1.5 or later.

Enabling eXtreme Scale agent on JDK Version 1.5 or above

The ObjectGrid agent can be enabled with a Java command line option with the following syntax:

```
-javaagent:jarpath[=options]
```

The *jarpath* value is the path to an eXtreme Scale runtime Java archive (JAR) file that contains eXtreme Scale agent class and supporting classes such as the `objectgrid.jar`, `wsobjectgrid.jar`, `ogclient.jar`, `wsogclient.jar`, and `ogagent.jar` files. Typically, in a stand-alone Java program or in a Java Platform, Enterprise Edition environment that is not running WebSphere Application Server, use the `objectgrid.jar` or `ogclient.jar` file. In a WebSphere Application Server or a multi-classloaders environment, you must use the `ogagent.jar` file in the Java command line agent option.

In addition to the agent JAR file, the `cglib.jar` file is required for inclusion in the bootstrap class path. In WebSphere Application Server, although the `cglib.jar` file might already be in the lib directory and included in application server class path, you must specify the `cglib.jar` file path in the class path property of the Java virtual machine (JVM) that is associated with the application server.

Provide the `ogagent.config` file in the class path or use agent options to specify additional information.

eXtreme Scale agent options

config
: Overrides the configuration file name.

include
: Specifies or overrides transformation domain definition that is the first part of the configuration file.

exclude
: Specifies or overrides the @Exclude definition.

fieldAccessEntity
: Specifies or overrides the @FieldAccessEntity definition.

trace
: Specifies a trace level. Levels can be ALL, CONFIG, FINE, FINER, FINEST, SEVERE, WARNING, INFO, and OFF.

trace.file
: Specifies the location of the trace file.

The semicolon (**;**) is used as a delimiter to separate each option. The comma (**,**) is used as a delimiter to separate each element within an option. The following example demonstrates the eXtreme Scale agent option for a Java program:

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myConfigFile;
include=includedPackage;exclude=excludedPackage;
fieldAccessEntity=package1,package2
```

`ogagent.config` file

The `ogagent.config` file is the designated eXtreme Scale agent configuration file name. If the file name is in the class path, the eXtreme Scale agent finds and parses the file. You can override the designated file name through the config option of eXtreme Scale agent. The following example shows how to specify the configuration file:

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
```

An eXtreme Scale agent configuration file has the following parts:

- **Transformation domain:** The transformation domain part is first in the configuration file. The transformation domain is a list of packages and classes that are included in the class transformation process. This transformation domain must include all classes that are field-access entity classes, and other classes that refer to these field-access entity classes. Field-access entity classes and those classes that refer to these field-access entity classes construct the transformation domain. If you plan to specify field-access entity classes in the @FieldAccessEntity part, then you do not need to include field-access entity classes here. The transformation domain must be complete. Otherwise, you might see a FieldAccessEntityNotInstrumentedException exception.

- **@Exclude:** The @Exclude token indicates that packages and classes listed after this token are excluded from the transformation domain.
- **@FieldAccessEntity:** The @FieldAccessEntity token indicates that packages and classes listed after this token are field-access Entity packages and classes. If no line exists after the @FieldAccessEntity token, then its equivalent is "No @FieldAccessEntity specified". The eXtreme Scale agent determines that there are no field-access Entity packages and classes defined. If there are lines after the @FieldAccessEntity token, then they represent the user-specified field-access Entity packages and classes. For example, "field-access entity domain". The field-access entity domain is a sub-domain of the transformation domain. Packages and classes that are listed in the field-access entity domain are a part of the transformation domain, even when they are not listed in the transformation domain. The @Exclude token, which lists packages and classes that are excluded from transformation, has no impact on the field-access Entity domain. When @FieldAccessEntity token is specified, all field-access entities must be in this field-access Entity domain. Otherwise, a FieldAccessEntityNotInstrumentedException exception might occur.

Example agent configuration file (ogagent.config)

```
################################
# The # indicates comment line
################################
# This is an ObjectGrid agent config file (the designated file name is ogagent.config) that can be found and parsed by the ObjectGrid agent
# if it is in classpath.
# If the file name is "ogagent.config" and in classpath, Java program runs with -javaagent:objectgridRoot/ogagent.jar will have
# ObjectGrid agent enabled.
# If the file name is not "ogagent.config" but in classpath, you can specify the file name in config option of ObjectGrid agent
#     -javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
# See comments below for more info regarding instrumentation setting override.

# The first part of the configuration is the list of packages and classes that should be included in transformation domain.
# The includes (packages/classes, construct the instrumentation doamin) should be in the beginning of the file.
com.testpackage
com.testClass

# Transformation domain: The above lines are packages/classes that construct the transformation domain.
# The system will process classes with name starting with above packages/classes for transformation.
#
# @Exclude token : Exclude from transformation domain.
# The @Exclude token indicates packages/classes after that line should be excluded from transformation domain.
# It is used when user want to exclude some packages/classes from above specified included packages
#
# @FieldAccessEntity token: Field-access Entity domain.
# The @FieldAccessEntity token indicates packages/classes after that line are field-access Entity packages/classes.
# If there is no lilne after the @FieldAccessEntity token, it is equivalent to "No @FieldAccessEntity specified".
# The runtime will consider the user does not specify any field-access Entity packages/classes.
# The "field-acces Entity domain" is a sub-domain of transformation domain.
#
# Packages/classes listed in the "field-access Entity domain" will always be part of transformation domain,
# even they are not listed in transformation domain.
# The @Exclude, which lists packages/classes excluded from transformation, has no impact on the "field-acces Entity domain".
# Note: When @FieldAccessEntity is specified, all field-access entities must be in this field-acces Entity domain,
#       otherwise, FieldAccessEntityNotInstrumentedException may occur.
#
# The default ObjectGrid agent config file name is ogagent.config
# The runtime will look for this file as a resource in classpath and process it.
# Users can override this designated ObjectGrid agent config file name via config option of agent.
#
# e.g.
# javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
#
# The instrumentation definition, including transformation domain, @Exclude, and  @FieldAccessEntity can be overriden individually
# by corresponding designated agent options.
# Designated agent options include:
#    include              -> used to override instrumentation domain definition that is the first part of the config file
#    exclude              -> used to override @Exclude definition
#    fieldAccessEntity    -> used to override @FieldAccessEntity definition
#
# Each agent option should be separated by ";"
# Within the agent option, the package or class should be seperated by ","
#
# The following is an example that does not override the config file name:
#    -javaagent:objectgridRoot/lib/objectgrid.jar=include=includedPackage;exclude=excludedPackage;fieldAccessEntity=package1,package2
#
################################

@Exclude
com.excludedPackage
com.excludedClass

@FieldAccessEntity
```

Performance consideration

For better performance, specify the transformation domain and field-access entity domain.

Entity query queues

Query queues allow applications to create a queue qualified by a query in the server-side or local eXtreme Scale over an entity. Entities from the query result are stored in this queue. Currently, query queue is only supported in a map that is using the pessimistic lock strategy.

A query queue is shared by multiple transactions/clients. After the query queue becomes empty, the entity query associated with this queue is re-run and new results are added to the queue. A query queue is uniquely identified by the entity query string and parameters. There is only one instance for each unique query queue in one ObjectGrid instance. See the EntityManager API documentation for additional information.

Query queue example

The following example shows how query queue can be used.

```
/**
 * Get a unassigned question type task
 */
private void getUnassignedQuestionTask() throws Exception {
    EntityManager em = og.getSession().getEntityManager();
    EntityTransaction tran = em.getTransaction();

    QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t
    WHERE t.type=?1 AND t.status=?2", Task.class);
    queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
    queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

    tran.begin();
    Task nextTask = (Task) queue.getNextEntity(10000);
    System.out.println("next task is " + nextTask);
    if (nextTask != null) {
        assignTask(em, nextTask);
    }
    tran.commit();
}
```

The previous example first creates a QueryQueue with a entity query string, "SELECT t FROM Task t WHERE t.type=?1 AND t.status=?2". Then it sets the QueryQueue object's parameters. This query queue represents all "unassigned" tasks of the type "question". The QueryQueue object is very similar to an entity Query object.

After the QueryQueue is created, an entity transaction is started and the getNextEntity method is invoked, which retrieves the next available entity with a timeout value set to 10 seconds. After the entity is retrieved, it is processed in the assignTask method. The assignTask modifies the Task entity instance and changes the status to "assigned" which effectively removes it from the queue since it no longer matches the QueryQueue's filter. Once assigned, the transaction is committed.

From this simple example, you can see a query queue is similar to an entity query. However, they differ in the following ways:

1. Entities in the query queue can be retrieved in an iterative manner. The user application decides the number of entities to be retrieved. For example, if QueryQueue.getNextEntity(timeout) is used, only one entity is retrieved, and if QueryQueue.getNextEntities(5, timeout) is used, 5 entities are retrieved. In a

distributed environment, the number of entities directly decides the number of bytes to be transferred from the server to client.

2. When an entity is retrieved from the query queue, a U lock is placed on the entity so no other transactions can access it.

Retrieve entities in a loop

You can retrieve entities in a loop. An example that illustrates how to get all the unassigned, question type tasks completed follows.

```
/**
 * Get all unassigned question type tasks
 */
private void getAllUnassignedQuestionTask() throws Exception {
    EntityManager em = og.getSession().getEntityManager();
    EntityTransaction tran = em.getTransaction();

    QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t WHERE
   t.type=?1 AND t.status=?2", Task.class);
    queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
    queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

    Task nextTask = null;

    do {
        tran.begin();
        nextTask = (Task) queue.getNextEntity(10000);
        if (nextTask != null) {
            System.out.println("next task is " + nextTask);
        }
        tran.commit();
    } while (nextTask != null);
}
```

If there are 10 unassigned question-type tasks in the entity map, you might expect that you will have 10 entities printed to the console. However, if this example is run, you will see the program never exits, which might be contrary to what you assumed.

When a query queue is created and the getNextEntity is called, the entity query associated with the queue is executed and the 10 results are populated into the queue. When getNextEntity is called, an entity is taken off the queue. After 10 getNextEntity calls are executed, the queue is empty. The entity query will automatically re-run. Since these 10 entities still exist and match the query queue's filter criteria, they are populated into the queue again.

If the following line is added after the println() statement, you will see only 10 entities printed.

```
em.remove(nextTask);
```

Query queues deployed to all partitions

In a distributed eXtreme Scale, a query queue can be created for one partition or all partitions. If a query queue is created for all partitions, there will be one query queue instance in each partition.

When a client tries to get the next entity using the QueryQueue.getNextEntity or QueryQueue.getNextEntities method, the client sends a request to one of the partitions. A client sends peek and pin requests to the server:

- With a peek request, the client sends a request to one partition and the server returns immediately. If there is an entity in the queue, the server sends a response with the entity; if there is not, the server sends a response with no entity. In either case, the server will return immediately.
- With a pin request, the client sends a request to one partition and the server waits until an entity is available. If there is an entity in the queue, the server sends a response with the entity immediately; if there is not, the server waits on the queue until either an entity is available or the request times out.

An example of how an entity is retrieved for a query queue which is deployed to all partitions (n) follows:

1. When a QueryQueue.getNextEntity or QueryQueue.getNextEntities method is called, the client picks a random partition number from 0 to n-1.
2. The client sends peek request to the random partition.
   - If an entity is available, the QueryQueue.getNextEntity or QueryQueue.getNextEntities method exits by returning the entity.
   - If an entity is not available and is not the last unvisited partition, the client sends a peek request to the next partition.
   - If an entity is not available and it is the last unvisited partition, the client instead sends a pin request.
   - If the pin request to the last partition times-out and there is still no data available, the client will make a last effort by sending peek request to all partitions serially one more round. Therefore, if any entity is available in the previous partitions, the client will be able to get it.

Subset entity and no-entity support

The method to create a QueryQueue object in the entity manager follows:

```
public QueryQueue createQueryQueue(String qlString, Class entityClass);
```

The result in the query queue should be projected to the object defined by the second parameter to the method, Class entityClass.

If this parameter is specified, the class must have the same entity name as specified in the query string. This is useful if you want to project an entity into a subset entity. If a null value is used as the entity class, then the result will not be projected. The value stored in the map will be in a entity tuple format.

Client-side key collision

In distributed eXtreme Scale environment, query queue is only supported for eXtreme Scale maps with pessimistic locking mode. Therefore, there is no near cache on the client side. However, a client could have data (key and value) in the transactional map. This potentially could lead to a key collision when an entity retrieved from the server share the same key as an entry already in the transactional map.

When a key collision happens, the eXtreme Scale client run time uses the following rule to either throw an exception or silently override the data.

1. If the collided key is the key of the entity specified in the entity query associated with the query queue, then an exception is thrown. In this case, the transaction is rolled back, and the U lock on this entity key will be released on the server side.

2. Otherwise, if the collided key is the key of the entity association, the data in the transactional map will be overridden without warning.

The key collision only happens when there is a data in the transactional map. In other words, it only happens when a getNextEntity or getNextEntities call is called in a transaction which has already been dirtied (a new data has been inserted or a data has been updated). If an application does not want a key collision happen, it should always call getNextEntity or getNextEntities in a transaction which has not been dirtied.

Client failures

After a client sends a getNextEntity or getNextEntities request to the server, the client could fail as follows:

1. The client sends a request to the server and then goes down.
2. The client gets one or more entities from the server and then goes down.

In the first case, the server discovers that the client is going down when it tries to send back the response to the client. In the second case, when the client gets one or more entities from the server, an X lock is placed on these entities. If the client goes down, the transaction will eventually time out, and the X lock will be released.

Query with ORDER BY clause

Generally, query queues do not honor the ORDER BY clause. If you call getNextEntity or getNextEntities from the query queue, there is no guarantee the entities are returned according to the order. The reason is that the entities cannot be ordered across partitions. In the case that the query queue is deployed to all partitions, when a getNextEntity or getNextEntities call is executed, a random partition is picked to process the request. Therefore, the order is not guaranteed.

ORDER BY is honored if a query queue is deployed to a single partition.

For more information see “EntityManager Query API” on page 93.

Related reference

API documentation: EntityManager interface

“EntityTransaction interface”
You can use the EntityTransaction interface to demarcate transactions.

EntityTransaction interface

You can use the EntityTransaction interface to demarcate transactions.

Purpose

To demarcate a transaction, you can use the EntityTransaction interface, which is associated with an entity manager instance. Use the EntityManager.getTransaction method to retrieve the EntityTransaction instance for the entity manager. Each EntityManager and EntityTransaction instance are associated with the Session. You can demarcate transactions with either the EntityTransaction or Session. Methods on the EntityTransaction interface do not have any checked exceptions. Only runtime exceptions of type PersistenceException or its subclasses result.

For more information about the EntityTransaction interface, see the EntityTransaction interface in the API documentation.

Related concepts

"Caching objects and their relationships (EntityManager API)" on page 51
Most cache products use map-based APIs to store data as key-value pairs. The ObjectMap API and the dynamic cache in WebSphere Application Server, among others, use this approach. However, map-based APIs have limitations. The EntityManager API simplifies the interaction with the data grid by providing an easy way to declare and interact with a complex graph of related objects.

"Entity query queues" on page 79
Query queues allow applications to create a queue qualified by a query in the server-side or local eXtreme Scale over an entity. Entities from the query result are stored in this queue. Currently, query queue is only supported in a map that is using the pessimistic lock strategy.

"EntityManager in a distributed environment" on page 62
You can use EntityManager with a local ObjectGrid or in a distributed eXtreme Scale environment. The main difference is how you connect to this remote environment. After you establish a connection, there is no difference between using a Session object or using the EntityManager API.

Query API

WebSphere eXtreme Scale provides a flexible query engine for retrieving entities using the EntityManager API and Java objects using the ObjectQuery API.

WebSphere eXtreme Scale query capabilities

With the eXtreme Scale query engine, you can perform SELECT type queries over an entity or object-based schema using the eXtreme Scale query language.

This query language provides the following capabilities:

- Single and multi-valued results
- Aggregate functions
- Sorting and grouping
- Joins
- Conditional expressions with subqueries
- Named and positional parameters
- eXtreme Scale index use
- Path expression syntax for object navigation
- Pagination

Query interface

Use the query interface to control entity query execution.

Use the EntityManager.createQuery(String) method to create a Query. You can use each query instance multiple times with the EntityManager instance in which it was retrieved.

Each query result produces an entity, where the entity key is the row ID (of type long) and the entity value contains the field results of the SELECT clause. You can use each query result in subsequent queries.

The following methods are available on the com.ibm.websphere.objectgrid.em.Query interface.

public ObjectMap getResultMap()

The getResultMap method runs a SELECT query and returns the results in an ObjectMap object with the results in query-specified order. The resulting ObjectMap is valid only for the current transaction.

The map key is the result number, of type long, starting at 1. The map value is of type com.ibm.websphere.projector.Tuple where each attribute and association is named based on its ordinal position within the select clause of the query. Use the method to retrieve the EntityMetadata for the Tuple object that is stored within the map.

The getResultMap method is the fastest method for retrieving query result data where multiple results can exist. You can retrieve the name of the resulting entity using the ObjectMap.getEntityMetadata() and EntityMetadata.getName() methods.

Example: The following query returns two rows.

```
String ql = SELECT e.name, e.id, d from Employee e join e.dept d WHERE d.number=5
 Query q = em.createQuery(ql);
 ObjectMap resultMap = q.getResultMap();
 long rowID = 1; // starts with index 1
 Tuple tResult = (Tuple) resultMap.get(new Long(rowID));
 while(tResult != null) {
     // The first attribute is name and has an attribute name of 1
     // But has an ordinal position of 0.
     String name = (String)tResult.getAttribute(0);
     Integer id = (String)tResult.getAttribute(1);

     // Dept is an association with a name of 3, but
     // an ordinal position of 0 since it's the first association.
     // The association is always a OneToOne relationship,
     // so there is only one key.
     Tuple deptKey = tResult.getAssociation(0,0);
     ...
     ++rowID;
     tResult = (Tuple) resultMap.get(new Long(rowID));

 }
```

public Iterator getResultIterator

The getResultIterator method runs a SELECT query and returns the query results using an Iterator where each result is either an Object for a single-valued query, or an Object array for a multiple-valued query. The values in the Object[] result are stored in query order. The result Iterator is valid for the current transaction only.

This method is preferred for retrieving query results within the EntityManager context. You can use the optional setResultEntityName(String) method to name the resulting entity so that it can be used in further queries.

Example: The following query returns two rows.

```
String ql = SELECT e.name, e.id, e.dept from Employee e WHERE e.dept.number=5
 Query q = em.createQuery(ql);
 Iterator results = q.getResultIterator();
 while(results.hasNext()) {
     Object[] curEmp = (Object[]) results.next();
     String name = (String) curEmp[0];
     Integer id = (Integer) curEmp[1];
     Dept d = (Dept) curEmp[2];
     ...
 }
```

public Iterator getResultIterator(Class resultType)

The getResultIterator(Class resultType) method runs a SELECT query and returns the query results using an entity Iterator. The entity type is determined by the resultType parameter. The result Iterator is valid only for the current transaction.

Use this method when you want to use the EntityManager APIs to access the resulting entities.

Example: The following query returns all of the employees and the department to which they belong for one division, ordering by salary. To print out the five employees with the highest salaries and then select work with employees from only one department in the same working set, use the following code.

```
String string_ql = "SELECT e.name, e.id, e.dept from Employee e WHERE
  e.dept.division='Manufacturing' ORDER BY e.salary DESC";
Query query1 = em.createQuery(string_ql);
query1.setResultEntityName("AllEmployees");
Iterator results1 = query1.getResultIterator(EmployeeResult.class);
int curEmployee = 0;
System.out.println("Highest paid employees");
while (results1.hasNext() && curEmployee++ < 5) {
 EmployeeResult curEmp = (EmployeeResult) results1.next();
 System.out.println(curEmp);
 // Remove the employee from the resultset.
 em.remove(curEmp);
}

// Flush the changes to the result map.
em.flush();

// Run a query against the local working set without the employees we
// removed
String string_q2 = "SELECT e.name, e.id, e.dept from AllEmployees e
  WHERE e.dept.name='Hardware'";
Query query2 = em.createQuery(string_q2);
Iterator results2 = query2.getResultIterator(EmployeeResult.class);
System.out.println("Subset list of Employees");
while (results2.hasNext()) {
 EmployeeResult curEmp = (EmployeeResult) results2.next();
 System.out.println(curEmp);
}
```

public Object getSingleResult

The getSingleResult method runs a SELECT query that returns a single result.

If the SELECT clause has more than one field defined, then the result is an object array, where each element in the array is based on its ordinal position within the SELECT clause of the query.

```
String ql = SELECT e from Employee e WHERE e.id=100"
 Employee e = em.createQuery(ql).getSingleResult();

 String ql = SELECT e.name, e.dept from Employee e WHERE e.id=100"
 Object[] empData = em.createQuery(ql).getSingleResult();
 String empName= (String) empData[0];
 Department empDept = (Department) empData[1];
```

public Query setResultEntityName(String entityName)

The setResultEntityName(String entityName) method specifies the name of the query result entity.

Each time the getResultIterator or getResultMap methods are invoked, an entity with an ObjectMap is dynamically created to hold the results of the query. If the entity is not specified, or null, the entity and ObjectMap name are automatically generated.

Because all query results are available for the duration of a transaction, a query name cannot be reused in a single transaction.

public Query setPartition(int partitionId)

Set the partition to where the query routes.

This method is required if the maps in the query are partitioned and if the entity manager does not have affinity to a single schema root entity partition.

Use the PartitionManager Interface to determine the number of partitions for the backing map of a given entity.

The following table provides descriptions of the other methods that are available through the query interface.

Table 9. Other methods.

Method	**Result**
public Query setMaxResults(int maxResult)	Set the maximum number of results to retrieve.
public Query setFirstResult(int startPosition)	Set the position of the first result to retrieve.
public Query setParameter(String name, Object value)	Bind an argument to a named parameter.
public Query setParameter(int position, Object value)	Bind an argument to a positional parameter.
public Query setFlushMode(FlushModeType flushMode)	Set the flush mode type to be used when the query runs, overriding the flush mode type set on the EntityManager.

eXtreme Scale query elements

With the eXtreme Scale query engine, you can use a single query language for searching the eXtreme Scale cache. This query language can query Java objects that are stored in ObjectMap objects and Entity objects. Use the following syntax for creating a query string.

An eXtreme Scale query is a string that contains the following elements:
- A SELECT clause that specifies the objects or values to return.
- A FROM clause that names the object collections.
- An optional WHERE clause that contains search predicates over the collections.
- An optional GROUP BY and HAVING clause (see eXtreme Scale query aggregation functions).
- An optional ORDER BY clause that specifies the ordering of the result collection.

Collections of Java objects are identified in queries through the use of their name in the query FROM clause.

The elements of query language are discussed in more detail in the following related topics:

- “ObjectGrid query Backus-Naur Form” on page 105 syntax
- “Reference for eXtreme Scale queries” on page 97

The following topics describe the means to use the Query API:

- “EntityManager Query API” on page 93
- “Using the ObjectQuery API” on page 88

Querying data in multiple time zones

In a distributed scenario, queries actually run on servers. When querying data with predicates of type calendar, java.util.Date and timestamp, the specified date time value in a query is based on the local time zone of the server.

In a single time-zone system where all clients and servers run on same time zone, you do not need to consider issues related to predicate types with calendar, java.util.Date and timestamp. However, when clients and servers are in different time zones, the specified date time value in queries is based on the server time zone and may return unwanted data back to client. Without knowing the server time zone, the specified date time value is meaningless. So the specified date time value should consider the time zone offset difference between the target time zone and the server time zone.

Time zone offset

For example, assume that a client is in [GMT-0] time zone and the server is in [GMT-6] time zone. The server time zone is 6 hours behind the client. The client would like to run the following query:

```
SELECT e FROM Employee e WHERE e.birthDate='1999-12-31 06:00:00'
```

Assuming the entity Employee has a birthDate attribute that is of type java.util.Date, the client is in [GMT-0] time zone and wants to retrieve Employees with birthDate value as '1999-12-31 06:00:00 [GMT-0]' based on its time zone.

The query will run on the server and the birthDate value used by the query engine will be '1999-12-31 06:00:00 [GMT-6]' that equals to '1999-12-31 12:00:00 [GMT-0]'. Employees with birthDate value equal to '1999-12-31 12:00:00 [GMT-0]' will be returned to the client. Thus, the client will not get wanted Employees with birthDate value '1999-12-31 06:00:00 [GMT-0]'.

The problem described occurs because of the time zone difference between client and server. To solve this problem, one approach is to calculate the time zone offset between client and server and apply the time zone offset on the target date time value in the query. In the previous query example, the time zone offset is -6 hours, and the adjusted birthDate predicate should be “birthDate='1999-12-31 00:00:00'” if the client intends to retrieve Employees with birthDate value '12-31 06:00:00 [GMT-0]'. With the adjusted birthDate value, the server will use '1999-12-31 00:00:00 [GMT-6]' that equals to target value '12-31 06:00:00 [GMT-0]', and the required Employees will be returned to the client.

Distributed deployment in multiple time zones

If the distributed eXtreme Scale grid is deployed into multiple ObjectGrid servers in various time zones, the adjusting time zone offset approach will not work because the client will not know which server will run the query and thus cannot determine the time zone offset to use. The only solution is to use suffix ‘Z' (not case sensitive) on JDBC date and time escape format to indicate using GMT time zone based date time value. The suffix ‘Z' (not case sensitive) indicates to use GMT time zone based date time value. Without the suffix ‘Z', the local time zone based date time value will be used in the process that runs the query.

The following query is equivalent to the previous example, but uses the suffix ‘Z' instead:

```
SELECT e FROM Employee e WHERE e.birthDate='1999-12-31 06:00:00Z'
```

The query should find Employees with birthDate value ‘1999-12-31 06:00:00'. The suffix ‘Z' indicates the specified birthDate value is GMT time zone based, so the GMT time zone based birthDate value ‘1999-12-31 06:00:00 [GMT-0]' will be used by the query engine for matching criteria value. Employees with birthDate attribute value equal to this GMT based birthDate value ‘1999-12-31 06:00:00 [GMT-0]' will be included in query result. Using the suffix ‘Z' on JDBC date time escape format in any query is crucial to make applications time zone safe. Without this approach, the date time value is server time zone based and is meaningless from the client perspective when clients and servers are in different time zones.

For more information, see the topic on inserting data for different time zones in the *Product Overview*.

Using the ObjectQuery API

The ObjectQuery API provides methods for querying data in the ObjectGrid that is stored using the ObjectMap API. When a schema is defined in the ObjectGrid instance, the ObjectQuery API can be used to create and run queries over the heterogeneous objects stored in the object maps.

Query and object maps

You can use an enhanced query capability for objects that are stored using the ObjectMap API. These queries allow retrieval of objects using non-key attributes and performs simple aggregations such as sum, avg, min, and max against all the data that matches a query. Applications can construct a query using the Session.createObjectQuery method. This method returns an ObjectQuery object which can then be interrogated to obtain the query results. The query object also allows the query to be customized before running the query. The query is run automatically when any method returning the result is called.

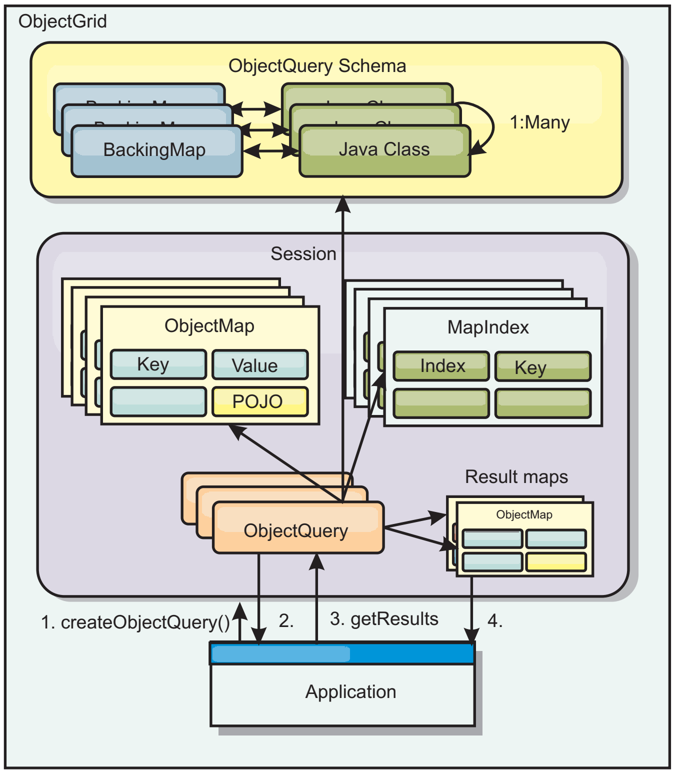

Figure 1. The interaction of the query with the ObjectGrid object maps and how a schema is defined for classes and associated with an ObjectGrid map

Defining an ObjectMap schema

Object maps are used to store objects in various forms and are largely unaware of the format. A schema must be defined in the ObjectGrid that defines the format of the data. A schema is composed of the following pieces:

- The type of object stored in the ObjectMap
- Relationships between ObjectMaps
- The method for which each query should access the data attributes in the objects (fields or property methods)
- The primary key attribute name in the object.

See Configuring an ObjectQuery schema for details.

For an example on creating a schema programmatically or using the ObjectGrid descriptor XML file, see the tutorial on the ObjectQuery in the *Product Overview*.

Querying objects with the ObjectQuery API

The ObjectQuery interface allows the querying of non-entity objects, which are heterogeneous objects that are stored directly in the ObjectGrid ObjectMaps. The ObjectQuery API provides an easy way to find ObjectMap objects without using the keyword and index mechanisms directly.

There are two methods for retrieving results from an ObjectQuery: getResultIterator and getResultMap.

Retrieving query results using getResultIterator

Query results are basically a list of attributes. Suppose the query was select a,b,c from X where y=z. This query returns a list of rows containing a, b and c. This list is actually stored in a transaction scoped Map, which means that you must associate an artificial key with each row and use an integer that increases with each row. This map is obtained using the ObjectQuery.getResultMap() method. You can access the elements of each row using code similar to the following:

```
ObjectQuery q = session.createQuery(
    "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");

  q.setParameter(1, "Claus");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id "
      + row[objectgrid: 0 ] + ", firstName: "
      + row[objectgrid: 1 ] + ", surname: "
      + row[objectgrid: 2 ]);
  }
```

Retrieving query results using getResultMap

Query results can also be retrieved using the result map directly. The following example shows a query retrieving specific parts of the matching Customers and demonstrates how to access the resulting rows. Notice that if you use the ObjectQuery object to access the data, then the generated long row identifier is hidden. The long row is only visible when using the ObjectMap to access the result.

When the transaction is completed this map disappears. The map is also only visible to the session used, that is, normally to just the thread that created it. The map uses a key of type Long which represents the row ID. The values stored in the map either are of type Object or Object[], where each element matches the type of the element in the select clause of query.

```
ObjectQuery q = em.createQuery(
      "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
  q.setParameter(1, "Claus");
  ObjectMap qmap = q.getResultMap();
  for(long rowId = 0; true; ++rowId)
  {
    Object[] row = (Object[]) qmap.get(new Long(rowId));
    if(row == null) break;
    System.out.println(" I Found a Claus with id " + row[0]
      + ", firstName: " + row[1]
      + ", surname: " + row[2]);
  }
```

For examples on using the ObjectQuery, see the tutorial on the ObjectQuery API in the *Product Overview*.

Configuring an ObjectQuery schema

ObjectQuery relies on schema or shape information to perform semantic checking and to evaluate path expressions. This section describes how to define the schema in XML or programmatically.

Defining the schema

The ObjectMap schema is defined in the ObjectGrid deployment descriptor XML or programmatically using the normal eXtreme Scale configuration techniques. For an example on how to create a schema, see "Configuring an ObjectQuery schema" on page 90

Schema information describes plain old Java objects (POJOs): which attributes they consist of and what types of attributes there might be, whether the attributes are primary key fields, single-valued or multi-valued relationships, or bidirectional relationships. Schema information directs ObjectQuery to use field access or property access.

Queryable attributes

When the schema is defined in the ObjectGrid, the objects in the schema are introspected using reflection to determine which attributes are available for querying. You can query the following attribute types:
- Java primitive types including wrappers
- java.lang.String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- java.util.Calendar
- byte[]
- java.lang.Byte[]
- char[]
- java.lang.Character[]
- J2SE enum

Embedded serializable types other than those stated previously can also be included in a query result, but cannot be included in the WHERE or FROM clause of the query. Serializable attributes are not navigable.

Attribute types can be excluded from the schema if the type is not serializable, the field or property is static, or the field is transient. Since all map objects must be serializable, the ObjectGrid only includes attributes that can be persisted from the object. Other objects are ignored.

Field attributes

When the schema is configured to access the object using fields, all serializable, non-transient fields are automatically incorporated into the schema. To select a field attribute in a query, use the field identifier name as it exists in the class definition.

All public, private, protected and package protected fields are included in the schema.

Property attributes

When the schema is configured to access the object using properties, all serializable methods that follow the JavaBeans property naming conventions will automatically be incorporated into the schema. To select a property attribute for the query, use the JavaBeans style property name conventions.

All public, private, protected and package protected properties are included in the schema.

In the following class, the following attributes are added to the schema: name, birthday, valid.

```
public class Person {
  public String getName(){}
  private java.util.Date getBirthday(){}
  boolean isValid(){}
  public NonSerializableObject getData(){}
}
```

When using a CopyMode of COPY_ON_WRITE, the query schema must always use property-based access. COPY_ON_WRITE creates proxy objects whenever objects are retrieved from the map and can only access those objects using property methods. Failure to do so will result in each query result being set to null.

Relationships

Each relationship must be explicitly defined in the schema configuration. The cardinality of the relationship is automatically determined by the type of the attribute. If the attribute implements the java.util.Collection interface, then the relationship is either a one-to-many or many-to-many relationship.

Unlike entity queries, attributes that refer to other cached objects must not store direct references to the object. References to other objects are serialized as part of the containing object's data. Instead, store the key to the related object.

For example, if there is a many-to-one relationship between a Customer and Order:

```
Incorrect. Storing an object reference.

public class Customer {
  String customerId;
  Collection<Order> orders;
}

public class Order {
  String orderId;
  Customer customer;
}
Correct. The key to the related object.

public class Customer {
  String customerId;
  Collection<String> orders;
}

public class Order {
  String orderId;
  String customer;
}
```

When a query is run that joins the two map objects together, the key will automatically be inflated. For example, the following query would return Customer objects:

```
SELECT c FROM Order o JOIN Customer c WHERE orderId=5
```

Using indexes

ObjectGrid uses index plugins to add indexes to maps. The query engine automatically incorporates any indexes that are defined on a schema map element of the type: com.ibm.websphere.objectgrid.plugins.index.HashIndex and the rangeIndex property is set to true. If the index type is not HashIndex and the rangeIndex property is not set to true, then the index is ignored by the query. See Object Query tutorial the ObjectQuery tutorial in the *Product Overview* for an example on how to add an index to the schema.

EntityManager Query API

The EntityManager API provides methods for querying data in the ObjectGrid that is stored using the EntityManager API. The EntityManager Query API is used to create and run queries over one or more entities defined in eXtreme Scale.

Query and ObjectMaps for entities

WebSphere Extended Deployment v6.1 introduced an enhanced query capability for entities stored in eXtreme Scale. These queries allow objects to be retrieved using non-key attributes and to perform simple aggregations such as the sum, average, minimum, and maximum against all the data that matches a query. Applications construct a query using the EntityManager.createQuery API. This returns a Query object and can then be interrogated to obtain the query results. The query object also allows the query to be customized before running the query. The query is run automatically when any method returning the result is called.

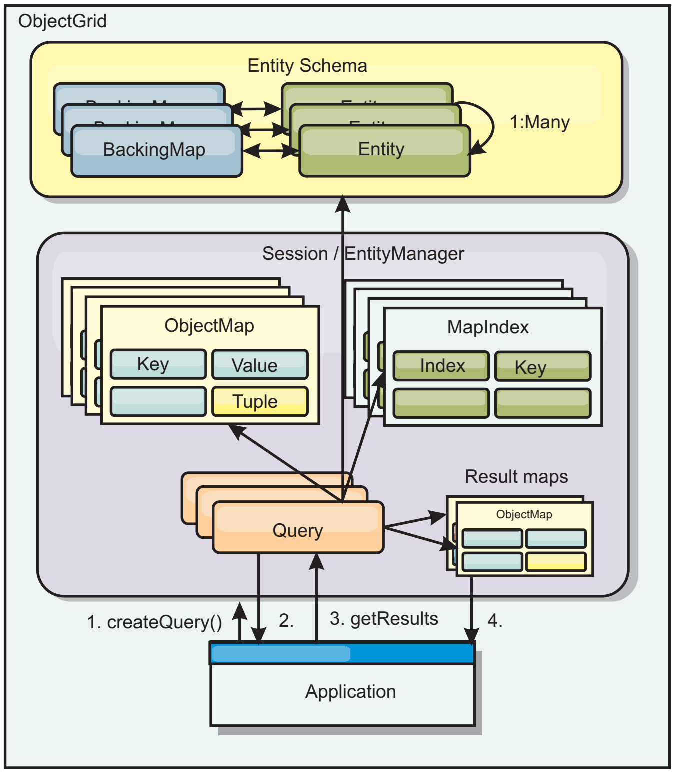

Figure 2. The interaction of the query with the ObjectGrid object maps and how the entity schema is defined and associated with an ObjectGrid map.

Retrieving query results using the getResultIterator method

Query results are a list of attributes. If the query was select a,b,c from X where y=z, then a list of rows containing a, b and c is returned. This list is stored in a transaction scoped Map, which means that you must associated an artificial key with each row and use an integer that increases with each row. This map is obtained using the Query.getResultMap method. The map has EntityMetaData, which describes each row in the Map associated with it. You can access the elements of each row using code similar to the following:

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");

  q.setParameter(1, "Claus");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id "  + row[objectgrid: 0 ]
      + ", firstName: " + row[objectgrid: 1 ]
      + ", surname: " + row[objectgrid: 2 ]);
  }
```

Retrieving query results using getResultMap

The following code shows the retrieval of specific parts of the matching Customers and shows how to access the resulting rows. If you use the Query object to access the data, then the generated long row identifier is hidden. The long is only visible when using the ObjectMap to access the result. When the transaction is completed, then this Map disappears. The Map is only visible to the Session used, that is, normally to just the thread that created it. The Map uses a Tuple for the key with a single attribute, a long with the row ID. The value is another tuple with an attribute for each column in the result set.

The following sample code demonstrates this:

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from
Customer c where c.surname=?1");
q.setParameter(1, "Claus");
ObjectMap qmap = q.getResultMap();
Tuple keyTuple = qmap.getEntityMetadata().getKeyMetadata().createTuple();
for(long i = 0; true; ++i)
{
  keyTuple.setAttribute(0, new Long(i));
  Tuple row = (Tuple)qmap.get(keyTuple);
  if(row == null) break;
  System.out.println(" I Found a Claus with id "  + row.getAttribute(0)
    + ", firstName: " + row.getAttribute(1)
    + ", surname: " + row.getAttribute(2));
}
```

Retrieving query results using an entity result iterator

The following code shows the query and the loop that retrieves each result row using the normal Map APIs. The key for the Map is a Tuple. So, construct one of the correct types using the createTuple method result in keyTuple. Try to retrieve all rows with rowIds from 0 onwards. When you get returns null (indicating key not found), then the loop finishes. Set the first attribute of keyTuple to be the long that you want to find. The value returned by get is also a Tuple with an attribute for each column in the query result. Then, pull each attribute from the value Tuple using getAttribute.

Following is the next code fragment:

```
Query q2 = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
q2.setResultEntityName("CustomerQueryResult");
q2.setParameter(1, "Claus");

Iterator iter2 = q2.getResultIterator(CustomerQueryResult.class);
while(iter2.hasNext())
{
  CustomerQueryResult row = (CustomerQueryResult)iter2.next();
  // firstName is the id not the firstName.
  System.out.println("Found a Claus with id " + row.id
    + ", firstName: " + row.firstName
    + ", surname: " + row.surname);
}

em.getTransaction().commit();
```

Specified is a ResultEntityName value on the query. This value tells the query engine that you want to project each row to a specific object, CustomerQueryResult in this case. The class follows:

```
@Entity
public class CustomerQueryResult {
 @Id long rowId;
 String id;
 String firstName;
 String surname;
};
```

In the first snippet, notice that the each query row is returned as a CustomerQueryResult object rather than an Object[]. The result columns of the query are projected to the CustomerQueryResult object. Projecting the result is slightly slower at run time but more readable. Query result Entities should not be registered with eXtreme Scale at startup. If the entities are registered, then a global Map with the same name is created, and the query fails with an error indicating duplicate Map name.

Simple queries with EntityManager

WebSphere eXtreme Scale comes with EntityManager query API.

The EntityManager query API is very similar to SQL other query engines that query over objects. A query is defined, then the result is retrieved from the query using various getResult methods.

The following examples refer to the entities used in the EntityManager tutorial in the Product Overview.

Running a simple query

In this example, customers with the surname of Claus are queried:

```
em.getTransaction().begin();

  Query q = em.createQuery("select c from Customer c where c.surname='Claus'");

  Iterator iter = q.getResultIterator();
  while(iter.hasNext())
  {
    Customer c = (Customer)iter.next();
    System.out.println("Found a claus with id " + c.id);
  }

  em.getTransaction().commit();
```

Using parameters

Since you want to find all customers with a surname of Claus, a parameter to specify the surname is used since you might may want to use this query more than once.

Positional Parameter Example

```
Query q = em.createQuery("select c from Customer c where c.surname=?1");
  q.setParameter(1, "Claus");
```

Using parameters is very important when the query is used more than once. The EntityManager needs to parse the query string and build a plan for the query, which is expensive. By using a parameter, the EntityManager caches the plan for the query, thereby reducing the time it takes to run a query.

Both positional and named parameters are used:

Named Parameter Example

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
  q.setParameter("name", "Claus");
```

Using an index to improve performance

If there are millions of customers, then the previous query needs to scan over all rows in the Customer Map. This is not very efficient. But eXtreme Scale provides a mechanism for defining indexes over individual attributes in an entity. The query automatically uses this index when appropriate, which can speed up queries dramatically.

You can specify which attributes to index very simply by using the @Index annotation on the entity attribute:

```
@Entity
public class Customer
{
  @Id String id;
  String firstName;
  @Index String surname;
  String address;
  String phoneNumber;
}
```

The EntityManager creates an appropriate ObjectGrid index for the surname attribute in the Customer entity and the query engine automatically uses the index, which greatly decreases the query time.

Using pagination to improve performance

If there are a million customers named Claus, then it is not likely that you would want to display a page displaying a million customers. It is more likely that you would want to display 10 or 25 customers at a time.

The Query setFirstResult and setMaxResults methods helps by only returning a subset of the results.

Pagination Example

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
  q.setParameter("name", "Claus");
  // Display the first page
  q.setFirstResult=1;
  q.setMaxResults=25;
  displayPage(q.getResultIterator());

  // Display the second page
  q.setFirstResult=26;
  displayPage(q.getResultIterator());
```

Reference for eXtreme Scale queries

WebSphere eXtreme Scale has its own language by which the user can query data.

ObjectGrid query FROM clause

The FROM clause specifies the collections of objects to which to apply the query. Each collection is identified either by an abstract schema name and an identification variable, called a range variable, or by a collection member declaration that identifies either a single or multi-valued relationship and an identification variable.

Conceptually, the semantics of the query is to first form a temporary collection of tuples, referred to as R. Tuples are composed of elements from the collections that are identified in the FROM clause. Each tuple contains one element from each of the collections in the FROM clause. All possible combinations are formed subject to the constraints that are imposed by the collection member declarations. If any schema name identifies a collection for which there are no records in the persistent store, then the temporary collection R is empty.

Examples using FROM

The DeptBean object contains records 10, 20 and 30. The EmpBean object contains records 1, 2 and 3 that are related to department 10 and records 4 and 5 that are related to department 20. Department 30 has no related employees.

```
FROM DeptBean d, EmpBean e
```

This clause forms a temporary collection R that contains 15 tuples.

```
FROM  DeptBean d,  DeptBean d1
```

This clause forms a temporary collection R that contains 9 tuples.

```
FROM  DeptBean d,  IN (d.emps) AS e
```

This clause forms a temporary collection R that contains 5 tuples. Department 30 is not in the R temporary collection because it contains no employees. Department 10 is contained in the R temporary collection three times and department 20 is contained in R twice.

Instead of using IN(d.emps) as e, you can use a JOIN predicate:

```
FROM DeptBean d JOIN d.emps as e
```

After forming the temporary collection, the search conditions of the WHERE clause are applied to the R temporary collection, yielding a new temporary collection R1. The ORDER BY and SELECT clauses are applied to R1 to yield the final result set.

An identification variable is a variable that is declared in the FROM clause using the IN operator or the optional AS operator.

```
FROM DeptBean AS d,  IN (d.emps) AS e
```

is equivalent to:

```
FROM DeptBean d, IN (d.emps) e
```

An identification variable that is declared to be an abstract schema name is called a range variable. In the previous query, "d" is a range variable. An identification variable that is declared to be a multi-valued path expression is called a collection member declaration. The "d" and "e" values in the previous example are collection member declarations.

An example of using a single-valued path expression in the FROM clause follows:

```
FROM EmpBean e, IN(e.dept.mgr) as m
```

ObjectGrid query SELECT clause

The syntax of the SELECT clause is illustrated in the following example:

```
SELECT { ALL | DISTINCT }  [ selection , ]*  selection
selection  ::= {single_valued_path_expression |
               identification_variable |
               OBJECT ( identification_variable) |
       aggregate_functions } [[ AS ] id ]
```

The SELECT clause consists of one or more of the following elements: a single identification variable that is defined in the FROM clause, a single-valued path expression that evaluates to object references or values, and an aggregate function. You can use the DISTINCT keyword to eliminate duplicate references.

A scalar-subselect is a subselect that returns a single value.

Examples using SELECT

Find all employees that earn more than the John employee:

```
SELECT OBJECT(e) FROM EmpBean ej, EmpBean eWHERE  ej.name = 'John'  and
e.salary > ej.salary
```

Find all departments that have one or more employees who earn less than 20000:

```
SELECT DISTINCT e.dept  FROM EmpBean e where e.salary < 20000
```

A query can have a path expression that evaluates to an arbitrary value:

```
SELECT e.dept.name  FROM EmpBean e where e.salary < 20000
```

The previous query returns a collection of name values for the departments that have employees who earn less than 20000.

A query can return an aggregate value:

```
SELECT  avg(e.salary)  FROM EmpBean e
```

A query that retrieves the names and object references for underpaid employees follows:

```
SELECT e.name as name, object(e) as emp from EmpBean e where e.salary <
50000
```

ObjectGrid query WHERE clause

The WHERE clause contains search conditions that are composed of the elements presented below. When a search condition evaluates to TRUE, the tuple is added to the result set.

ObjectGrid query literals

A string literal is enclosed in single quotes. A single quotation mark that occurs within a string literal is represented by two single quotes, for example: 'Tom''s'.

A numeric literal can be any of the following values:
- An exact value such as 57, -957, or +66
- Any value supported by Java long type
- A decimal literal such as 57.5 or -47.02
- An approximate numeric value such as 7E3 or -57.4E-2
- Float types must include the "F" qualifier, for example 1.0F
- Long types must include the "L" qualifier, for example 123L

Boolean literals are TRUE and FALSE.

Temporal literals follow JDBC escape syntax based on the type of attribute:
- java.util.Date: yyyy-mm-ss
- java.sql.Date: yyyy-mm-ss
- java.sql.Time: hh-mm-ss
- java.sql.Timestamp: yyyy-mm-dd hh:mm:ss.f...
- java.util.Calendar: yyyy-mm-dd hh:mm:ss.f...

Enum literals are expressed using Java enum literal syntax using the fully qualified enum class name.

ObjectGrid query input parameters

You can specify input parameters by either using an ordinal position or by using a variable name. Writing queries that use input parameters is strongly encouraged, because using input parameters increases performance by allowing the ObjectGrid to catch the query plan between running actions.

An input parameter can be any of the following types: Byte, Short, Integer, Long, Float, Double, BigDecimal, BigInteger, String, Boolean, Char, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar, a Java SE 5 enum, an Entity or POJO Object, or a binary data string in the form of Java byte[].

An input parameter must not have a NULL value. To search for the occurrence of a NULL value, use the NULL predicate.

Positional Parameters

Positional input parameters are defined by using question mark followed by a positive number:

`?[positive integer]`.

Positional input parameters are numbered starting at 1 and correspond to the arguments of the query; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

Example: `SELECT e FROM Employee e WHERE e.city = ?1 and e.salary >= ?2`

Named Parameters

Named input parameters are defined using a variable name in the format: :[parameter name].

Example: `SELECT e FROM Employee e WHERE e.city = :city and e.salary >= :salary`

ObjectGrid query BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values.

```
expression [NOT]  BETWEEN expression-2  AND expression-3
```

Example 1

```
e.salary BETWEEN 50000 AND 60000
```

is equivalent to:

```
e.salary >= 50000  AND e.salary <= 60000
```

Example 2

```
e.name NOT BETWEEN 'A' AND 'B'
```

is equivalent to:

```
e.name < 'A'  OR e.name > 'B'
```

ObjectGrid query IN predicate

The IN predicate compares a value to a set of values. You can use the IN predicate in one of two forms:

```
expression [NOT] IN ( subselect )expression [NOT] IN  ( value1, value2,
.... )
```

The ValueN value can either be a literal value or an input parameter. The expression cannot evaluate to a reference type.

Example 1

```
e.salary IN ( 10000, 15000 )
```

is equivalent to

```
( e.salary = 10000 OR  e.salary = 15000 )
```

Example 2

```
e.salary IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary = ANY   ( select  e1.salary from EmpBean e1 where e1.dept.deptno =
10)
```

Example 3

```
e.salary NOT IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno =
10)
```

is equivalent to

```
e.salary <> ALL    ( select  e1.salary from EmpBean e1 where e1.dept.deptno
= 10)
```

ObjectGrid query LIKE predicate

The LIKE predicate searches a string value for a certain pattern.

```
string-expression   [NOT] LIKE  pattern    [ ESCAPE escape-character ]
```

The pattern value is a string literal or parameter marker of type string in which the underscore (_) stands for any single character and percent (%) stands for any sequence of characters, including an empty sequence. Any other character stands for itself. The escape character can be used to search for character _ and %. The escape character can be specified as a string literal or as an input parameter.

If the string-expression is null, then the result is unknown.

If both string-expression and pattern are empty, then the result is true.

Example

```
'' LIKE '' is true
'' LIKE '%' is true
e.name LIKE '12%3' is true for '123' '12993' and false for '1234'
e.name LIKE 's_me' is true for 'some' and 'same', false for 'soome'
e.name LIKE '/_foo' escape '/' is true for '_foo', false for 'afoo'
e.name LIKE '//_foo' escape '/' is true for '/afoo' and for '/bfoo'
e.name LIKE '///_foo' escape '/' is true for '/_foo' but false for '/afoo'
```

ObjectGrid query NULL predicate

The NULL predicate tests for null values.

```
{single-valued-path-expression | input_parameter} IS [NOT] NULL
```

Example

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

ObjectGrid query EMPTY collection predicate

Use the EMPTY collection predicate to test for an empty collection.

To test if a multi-valued relationship is empty, use the following syntax:

```
collection-valued-path-expression IS [NOT]  EMPTY
```

Example

Empty collection predicate To find all the departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d.emps IS EMPTY
```

ObjectGrid query MEMBER OF predicate

The following expression tests whether the object reference that is specified by the single valued path expression or input parameter is a member of the designated collection. If the collection valued path expression designates an empty collection, then the value of the MEMBER OF expression is FALSE.

```
{ single-valued-path-expression | input_parameter } [ NOT ] MEMBER [ OF ]
collection-valued-path-expression
```

Example

Find employees that are not members of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

Find employees whose manager is a member of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d
WHERE e.dept.mgr MEMBER  OF d.emps  and d.deptno=?1
```

ObjectGrid query EXISTS predicate

The EXISTS predicate tests for the presence or absence of a condition that specified by a subselect.

```
EXISTS ( subselect )
```

The result of EXISTS is true if the subselect returns at least one value, otherwise the result is false.

To negate an EXISTS predicate, precede the predicate with the NOT logical operator.

Example

Return departments that have at least one employee that earns more than 1000000:

```
SELECT  OBJECT(d) FROM  DeptBean d
WHERE EXISTS ( SELECT  e  FROM IN (d.emps) e WHERE  e.salary > 1000000 )
```

Return departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d
WHERE NOT EXISTS  ( SELECT e FROM IN (d.emps) e)
```

You can also rewrite the previous query like in the following example:

```
SELECT OBJECT(d) FROM DeptBean d WHERE SIZE(d.emps)=0
```

ObjectGrid query ORDER BY clause

The ORDER BY clause specifies an ordering of the objects in the result collection. An example follows:

ORDER BY [order_element ,]* order_element order_element ::={ path-expression }[ASC | DESC]

The path expression must specify a single-valued field that is a primitive type of byte, short, int, long, float, double, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Character, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp and java.util.Calendar. The ASC order element specifies that the results are displayed in ascending order, which is the default. A DESC order element specifies that the results are displayed in descending order.

Example

Return department objects. Display the department numbers in decreasing order:

```
SELECT OBJECT(d) FROM DeptBean d ORDER BY d.deptno DESC
```

Return employee objects, sorted by department number and name:

```
SELECT OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC, e.name DESC
```

ObjectGrid query aggregation functions

Aggregation functions operate on a set of values to return a single scalar value. You can use these functions in the select and subselect methods. The following example illustrates an aggregation:

```
SELECT SUM (e.salary) FROM EmpBean e WHERE e.dept.deptno =20
```

This aggregation computes the total salary for department 20.

The aggregation functions are: AVG, COUNT, MAX, MIN, and SUM. The syntax of an aggregation function is illustrated in the following example:

```
aggregation-function ( [ ALL | DISTINCT ] expression )
```

or:

```
COUNT( [ ALL | DISTINCT ] identification-variable )
```

The DISTINCT option eliminates duplicate values before applying the function. The ALL option is the default option, and does not eliminate duplicate values. Null values are ignored in computing the aggregate function except when you use the COUNT(identification-variable) function, which returns a count of all the elements in the set.

Defining return type

The MAX and MIN functions can apply to any numeric, string or date-time data type and return the corresponding data type. The SUM and AVG functions take a numeric type as input. The AVG function returns a double type. The SUM function returns a long type if the input type is an integer type, except when the input is a Java BigInteger type, then the function returns a Java BigInteger type. The SUM function returns a double type if the input type is not an integer type, except when the input is a Java BigDecimal type, then the function returns a Java BigDecimal type. The COUNT function can take any data type except collections, and returns a long type.

When applied to an empty set, the SUM, AVG, MAX, and MIN functions can return a null value. The COUNT function returns zero (0) when it is applied to an empty set.

Using GROUP BY and HAVING clauses

The set of values that is used for the aggregate function is determined by the collection that results from the FROM and WHERE clause of the query. You can divide the set into groups and apply the aggregation function to each group. To perform this action, use a GROUP BY clause in the query. The GROUP BY clause defines grouping members, which comprise a list of path expressions. Each path expression specifies a field that is a primitive type of byte, short, int, long, float, double, boolean, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar or a Java SE 5 enum.

The following example illustrates the use of the GROUP BY clause in a query that computes the average salary for each department:

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

In division of a set into groups, a NULL value is considered equal to another NULL value.

Groups can be filtered using a HAVING clause that tests group properties before involving aggregate functions or grouping members. This filtering is similar to how the WHERE clause filters tuples (that is, records of the return collection values) from the FROM clause. An example of the HAVING clause follows:

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING COUNT(e) > 3 AND e.dept.deptno > 5
```

This query returns the average salary for departments that have more than three employees and the department number is greater than five.

You can use a HAVING clause without a GROUP BY clause. In this case, the entire set is treated as a single group, to which the HAVING clause is applied.

ObjectGrid query Backus-Naur Form

A summary of the ObjectGrid Query Backus-Naur Form (BNF) Notation follows.

Table 10. Key to BNF summary

Representation	Description
{...}	Grouping
[...]	Optional constructs
bold	Keywords
*	Zero or more
|	Alternates

```
ObjectGrid QL ::=select_clause from_clause [where_clause] [group_by_clause]
  [having_clause] [order_by_clause]

from_clause ::=FROM identification_variable_declaration
  [,identification_variable_declaration]*

identification_variable_declaration ::=collection_member_declaration |
  range_variable_declaration

collection_member_declaration ::=IN ( collection_valued_path_expression |
  single_valued_navigation) [AS] identifier | [LEFT [OUTER]
  | INNER] JOIN collection_valued_path_expression |
  single_valued_navigation [AS] identifier

range_variable_declaration ::=abstract_schema_name [AS] identifier

single_valued_path_expression ::={single_valued_navigation | identification_variable}.
  { state_field | state_field.value_object_attribute } | single_valued_navigation

single_valued_navigation ::=identification_variable.[ single_valued_association_field. ]*
  single_valued_association_field

collection_valued_path_expression ::=identification_variable.[
  single_valued_association_field. ]* collection_valued_association_field

select_clause ::= SELECT [DISTINCT] [ selection , ]* selection

selection ::= {single_valued_path_expression |identification_variable | OBJECT
  ( identification_variable) |aggregate_functions } [[ AS ] id ]
```

```
order_by_clause ::= ORDER BY [ {identification_variable.[ single_valued_association_field.
  ]*state_field} [ASC|DESC],]* {identification_variable.[
  single_valued_association_field. ]*state_field}[ASC|DESC]

where_clause ::= WHERE conditional_expression

conditional_expression ::= conditional_term | conditional_expression OR conditional_term

conditional_term ::= conditional_factor | conditional_term AND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary ::= simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression | like_expression |
  in_expression | null_comparison_expression | empty_collection_comparison_expression |
  exists_expression | collection_member_expression

between_expression ::= numeric_expression [NOT] BETWEEN numeric_expression
  AND numeric_expression | string_expression [NOT] BETWEEN
  string_expression AND string_expression | datetime_expression [NOT]
  BETWEEN datetime_expression AND datetime_expression

in_expression ::= identification_variable.[ single_valued_association_field. ]state_field
  [*NOT] IN { (subselect) | ( atom ,]* atom) }

atom ::= { string_literal | numeric_literal | input_parameter }

like_expression ::=string_expression [NOT] LIKE {string_literal | input_parameter}
  [ESCAPE {string_literal | input_parameter}]

null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS
  [ NOT ] NULL

empty_collection_comparison_expression ::= collection_valued_path_expression IS
  [NOT] EMPTY

collection_member_expression ::={ ssingle_valued_path_expression | input_parameter }[
  NOT ] MEMBER [ OF ]collection_valued_path_expression

exists_expression ::= EXISTS {(subselect)}

subselect ::= SELECT [{ ALL | DISTINCT }] subselection from_clause
  [where_clause] [group_by_clause] [having_clause]

subselection ::= {single_valued_path_expression |identification_variable |
  aggregate_functions }

group_by_clause ::= GROUP BY[single_valued_path_expression,]*
  single_valued_path_expression

having_clause ::= HAVING conditional_expression

comparison_expression ::= numeric_ expression comparison_operator { numeric_expression
  | {SOME | ANY | ALL} (subselect) } | string_expression
  comparison_operator {

string_expression | {SOME | ANY | ALL}(subselect) } |

datetime_expression comparison_operator {

datetime_expression {SOME | ANY | ALL}(subselect) } |

boolean_expression {=|<>} {

boolean_expression {SOME | ANY | ALL}(subselect) } |

entity_expression {=|<>} {

entity_expression {SOME| ANY | ALL}(subselect) }

comparison_operator ::= = | > | >= | < | <= | <>

string_expression ::= string_primary | (subselect)

string_primary ::=state_field_path_expression |string_literal | input_parameter |
  functions_returning_strings

datetime_expression ::= datetime_primary |(subselect)

datetime_primary ::=state_field_path_expression | string_literal | long_literal
  | input_parameter | functions_returning_datetime

boolean_expression ::= boolean_primary |(subselect)

boolean_primary ::=state_field_path_expression | boolean_literal | input_parameter

entity_expression ::=single_valued_association_path_expression |
  identification_variable | input_parameter

numeric_expression ::= simple_numeric_expression |(subselect)

simple_numeric_expression ::= numeric_term | numeric_expression {+|-} numeric_term

numeric_term ::= numeric_factor | numeric_term {*|/} numeric_factor

numeric_factor ::= {+|-} numeric_primary
```

```
numeric_primary ::= single_valued_path_expression | numeric_literal |
  ( numeric_expression ) | input_parameter | functions
aggregate_functions :=
AVG([ALL|DISTINCT] identification_variable.
  [ single_valued_association_field. ]*state_field) |
COUNT([ALL|DISTINCT] {single_valued_path_expression |
  identification_variable}) |
MAX([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field) |
MIN([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field) |
SUM([ALL|DISTINCT] identification_variable.[
  single_valued_association_field. ]*state_field)
functions ::=
ABS (simple_numeric_expression) |
CONCAT (string_primary , string_primary) |
LOWER (string_primary) |
LENGTH(string_primary) |
LOCATE(string_primary, string_primary [, simple_numeric_expression]) |
MOD (simple_numeric_expression, simple_numeric_expression) |
SIZE (collection_valued_path_expression) |
SQRT (simple_numeric_expression) |
SUBSTRING (string_primary, simple_numeric_expression[, simple_numeric_expression]) |
UPPER (string_primary) |
TRIM ([[LEADING | TRAILING | BOTH] [trim_character]
  FROM] string_primary)
```

Query performance tuning

To tune the performance of your queries, use the following techniques and tips.

Using parameters

When a query runs, the query string must be parsed and a plan developed to run the query, both of which can be costly.WebSphere eXtreme Scale caches query plans by the query string. Since the cache is a finite size, it is important to reuse query strings whenever possible. Using named or positional parameters also helps performance by fostering query plan reuse.

```
Positional Parameter Example Query q = em.createQuery("select c from
Customer c where c.surname=?1");   q.setParameter(1, "Claus");
```

Using indexes

Proper indexing on a map might have a significant impact on query performance, even though indexing has some overhead on overall map performance. Without indexing on object attributes involved in queries, the query engine performs a table scan for each attribute. The table scan is the most expensive operation during a query run. Indexing on object attributes that are involved in queries allow the query engine to avoid an unnecessary table scan, improving the overall query performance. If the application is designed to use query intensively on a read-most map, configure indexes for object attributes that are involved in the query. If the map is mostly updated, then you must balance between query performance improvement and indexing overhead on the map. See “Indexing” on page 118 for more information.

When plain old Java objects (POJO) are stored in a map, proper indexing can avoid a Java reflection. In the following example, query replaces the WHERE clause with range index search, if the budget field has an index built over it. Otherwise, query scans the entire map and evaluates the WHERE clause by first getting the budget using Java reflection and then comparing the budget with the value 50000:

```
SELECT d FROM DeptBean d WHERE d.budget=50000
```

See “Query plan” for details on how to best tune individual queries and how different syntax, object models and indexes can affect query performance.

Using pagination

In client-server environments, the query engine transports the entire result map to the client. The data that is returned should be divided into reasonable chunks. The EntityManager Query and ObjectMap ObjectQuery interfaces both support the setFirstResult and setMaxResults methods that allow the query to return a subset of the results.

Return primitive values instead of entities

With the EntityManager Query API, entities are returned as query parameters. The query engine currently returns the keys for these entities to the client. When the client iterates over these entities using the Iterator from the getResultIterator method, each entity is automatically inflated and managed as if it were created with the find method on the EntityManager interface. The entire entity graph is built from the entity ObjectMap on the client. The entity value attributes and any related entities are eagerly resolved.

To avoid building the costly graph, modify the query to return the individual attributes with path navigation.

For example:

```
// Returns an entity
SELECT p FROM Person p
// Returns attributes SELECT p.name, p.address.street, p.address.city, p.gender FROM Person p
```

Query plan

All eXtreme Scale queries have a query plan. The plan describes how the query engine will interact with ObjectMaps and indexes. Display the query plan to determine if the query string or indexes are being used appropriately. The query plan can also be used to explore the differences that subtle changes in a query string make in the way eXtreme Scale runs a query.

The query plan can be viewed one of two ways:

- EntityManager Query or ObjectQuery getPlan API methods
- ObjectGrid diagnostic trace

getPlan method

The getPlan method on the ObjectQuery and Query interfaces return a String which describes the query plan. This string can be displayed to standard output or a log to display a query plan. Note: In a distributed environment, the getPlan method does not run against the server and will not reflect any defined indexes. To view the plan, use an agent to view the plan on the server.

Query plan trace

The query plan can be displayed using ObjectGrid trace. To enable query plan trace, use the following trace specification:

QueryEnginePlan=debug=enabled

See "Logs and trace" on page 271 for details on how to enable trace and locate the trace log files.

Query plan examples

Query plan uses the word for to indicate that the query is iterating through an ObjectMap collection or through a derived collection such as: q2.getEmps(), q2.dept or a temporary collection returned by an inner loop. If the collection is from an ObjectMap, the query plan shows whether a sequential scan (denoted by INDEX SCAN), unique or non-unique index is used. Query plan uses a filter string to list the condition expressions applied to a collection.

A Cartesian product is not commonly used in object query. The following query scans the entire EmpBean map in the outer loop and scans the entire DeptBean map in the inner loop:

```
SELECT e, d FROM EmpBean e, DeptBean d

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in DeptBean ObjectMap using INDEX SCAN
   returning new Tuple( q2,  q3  )
```

The following query retrieves all employee names from a particular department by sequentially scanning the EmpBean map to get an employee object. From the employee object, the query navigates to its department object and applies the d.no=1 filter. In this example, each employee has only one department object reference, so the inner loop runs once:

```
SELECT e.name FROM EmpBean e JOIN e.dept d WHERE d.no=1

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter ( q3.getNo() = 1 )
   returning new Tuple( q2.name  )
```

The following query is equivalent to the previous query. However, the query below performs better because it first narrows the result down to one department object by using the unique index that is defined over the DeptBean primary key field number. From the department object, the query navigates to its employee objects to get their names:

```
SELECT e.name FROM DeptBean d JOIN d.emps e WHERE d.no=1

Plan trace:

for q2 in DeptBean ObjectMap using UNIQUE INDEX key=(1)
     for q3 in  q2.getEmps()
   returning new Tuple( q3.name  )
```

The following query finds all the employees that work for development or sales. The query scans the entire EmpBean map and performs additional filtering by evaluating the expressions: d.name = 'Sales' or d.name='Dev'

```
SELECT e FROM EmpBean e, in (e.dept) d WHERE d.name = 'Sales'
 or d.name='Dev'

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
     for q3 in  q2.dept
      filter (( q3.getName() = Sales ) OR ( q3.getName() = Dev ) )
   returning new Tuple( q2  )
```

The following query is equivalent to the previous query, but this query runs a different query plan and uses the range index built over the field name. In general, this query performs better because the index over the name field is used for narrowing down the department objects, which run quickly if only a few departments are development or sales.

```
SELECT e FROM DeptBean d, in(d.emps) e WHERE d.name='Dev' or d.name='Sales'

Plan trace:

IteratorUnionIndex of

     for q2 in DeptBean ObjectMap using INDEX on name = (Dev)
       for q3 in  q2.getEmps()


     for q2 in DeptBean ObjectMap using INDEX on name = (Sales)
       for q3 in  q2.getEmps()
```

The following query finds departments that do not have any employees:

```
SELECT d FROM DeptBean d WHERE NOT EXISTS(select e from d.emps e)

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( NOT  EXISTS (    correlated collection defined as

       for q3 in  q2.getEmps()
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

The following query is equivalent to the previous query but uses the SIZE scalar function. This query has similar performance but is easier to write.

```
SELECT d FROM DeptBean d WHERE SIZE(d.emps)=0
for q2 in DeptBean ObjectMap using INDEX SCAN
    filter (SIZE( q2.getEmps()) = 0 )
   returning new Tuple( q2  )
```

The following example is another way of writing the same query as the previous query with similar performance, but this query is easier to write as well:

```
SELECT d FROM DeptBean d WHERE d.emps is EMPTY

Plan trace:

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( q2.getEmps() IS EMPTY  )
   returning new Tuple( q2  )
```

The following query finds any employees with a home address matching at least one of the addresses of the employee whose name equals the value of the parameter. The inner loop has no dependency on the outer loop. The query runs the inner loop once.

```
SELECT e FROM EmpBean e WHERE e.home =  any (SELECT e1.home FROM EmpBean e1
 WHERE e1.name=?1)
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.home =ANY     temp collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = ( ?1)
       returning new Tuple( q3.home      )
 )
   returning new Tuple( q2  )
```

The following query is equivalent to the previous query, but has a correlated subquery; also, the inner loop runs repeatedly.

```
SELECT e FROM EmpBean e WHERE EXISTS(SELECT e1 FROM EmpBean e1 WHERE
 e.home=e1.home and e1.name=?1)

Plan trace:

for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( EXISTS (    correlated collection defined as

       for q3 in EmpBean ObjectMap using INDEX on name = (?1)
        filter ( q2.home =  q3.home )
       returning new Tuple( q3      )

   returning new Tuple( q2  )
```

Query optimization using indexes

Defining and using indexes properly can significantly improve query performance.

WebSphere eXtreme Scale queries can use built-in HashIndex plug-ins to improve performance of queries. Indexes can be defined on entity or object attributes. The query engine will automatically use the defined indexes if its WHERE clause uses one of the following strings:

- A comparison expression with the following operators: =, <, >, <= or >= (any comparison expressions except not equals <>)
- A BETWEEN expression
- Operands of the expressions are constants or simple terms

Requirements

Indexes have the following requirements when used by Query:

- All indexes must use the built-in HashIndex plug-in.
- All indexes must be statically defined. Dynamic indexes are not supported.
- The @Index annotation may be used to automatically create static HashIndex plug-ins.
- All single-attribute indexes must have the RangeIndex property set to true.
- All composite indexes must have the RangeIndex property set to false.
- All association (relationship) indexes must have the RangeIndex property set to false.

For a more efficient way to search for cached objects, see “Composite HashIndex” on page 123

Using hints to choose an index

An index can be manually selected using the setHint method on the Query and ObjectQuery interfaces with the HINT_USEINDEX constant. This can be helpful when optimizing a query to use the best performing index.

Query examples that use attribute indexes

The following examples use simple terms: e.empid, e.name, e.salary, d.name, d.budget and e.isManager. The examples assume that indexes are defined over the name, salary and budget fields of an entity or value object. The empid field is a primary key and isManager has no index defined.

The following query uses both indexes over the fields of name and salary. It returns all employees with names that equal the value of the first parameter or a salary equal to the value of the second parameter:

```
SELECT e FROM EmpBean e where e.name=?1 or e.salary=?2
```

The following query uses both indexes over the fields of name and budget. The query returns all departments named 'DEV' with a budget that is greater than 2000.

```
SELECT d FROM DeptBean dwhere d.name='DEV' and d.budget>2000
```

The following query returns all employees with a salary greater than 3000 and with an isManager flag value that equals the value of the parameter. The query uses the index that is defined over the salary field and performs additional filtering by evaluating the comparison expression: `e.isManager=?1`.

```
SELECT e FROM EmpBean e where e.salary>3000 and e.isManager=?1
```

The following query finds all employees who earn more than the first parameter, or any employee that is a manager. Although the salary field has an index defined, query scans the built-in index that is built over the primary keys of the EmpBean field and evaluates the expression: `e.salary>?1` or `e.isManager=TRUE`.

```
SELECT e FROM EmpBean e WHERE e.salary>?1 or e.isManager=TRUE
```

The following query returns employees with a name that contains the letter a. Although the name field has an index defined, query does not use the index because the name field is used in the LIKE expression.

```
SELECT e FROM EmpBean e WHERE e.name LIKE '%a%'
```

The following query finds all employees with a name that is not "Smith". Although the name field has an index defined, query does not use the index because the query uses the not equals (<>) comparison operator.

```
SELECT e FROM EmpBean e where e.name<>'Smith'
```

The following query finds all departments with a budget less than the value of the parameter, and with an employee salary greater than 3000. The query uses an index for the salary, but it does not use an index for the budget because dept.budget is not a simple term. The dept objects are derived from collection e. You do not need to use the budget index to look for dept objects.

```
SELECT dept from EmpBean e, in (e.dept) dept where e.salary>3000 and
dept.budget<?
```

The following query finds all employees with a salary greater than the salary of the employees that have the empid of 1, 2, and 3. The index salary is not used because the comparison involves a subquery. The empid is a primary key, however, and is used for a unique index search because all the primary keys have a built-in index defined.

```
SELECT e FROM EmpBean e WHERE e.salary > ALL (SELECT e1.salary    FROM
EmpBean e1 WHERE e1.empid=1 or e1.empid =2 or e1.empid=99)
```

To check if the index is being used by the query, you can view the "Query plan" on page 108. Here is an example query plan for the previous query:

```
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.salary >ALL     temp collection defined as
       IteratorUnionIndex of

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(1)
        )

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(2)
        )

         for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(99)
        )
       returning new Tuple( q3.salary )
   returning new Tuple( q2  )

for q2 in EmpBean ObjectMap using RANGE INDEX on salary with range(3000,)
     for q3 in  q2.dept
      filter ( q3.budget <  ?1 )
   returning new Tuple( q3  )
```

Indexing attributes

Indexes can be defined over any single attribute type with the constraints previously defined.

Defining entity indexes using @Index

To define an index on an entity, simply define an annotation:

Entities using annotations

```
  @Entity
  public class Employee {
  @Id int empid;
  @Index String name
  @Index double salary
  @ManyToOne Department dept;
}
@Entity
  public class Department {
  @Id int deptid;
  @Index String name;
  @Index double budget;
  boolean isManager;
  @OneToMany Collection<Employee> employees;
  }
```

With XML

Indexes can also be defined using XML:

Entities without annotations

```
public class Employee {
int empid;
String name
double salary
Department dept;
}

public class Department {
int deptid;
String name;
double budget;
boolean isManager;
Collection employees;
}
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="DepartmentGrid" entityMetadataXMLFile="entity.xml>
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Emp">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.salary"/>
  <property name="AttributeName" type="java.lang.String" value="salary"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="Dept">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.budget"/>
  <property name="AttributeName" type="java.lang.String" value="budget"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>Department entities</description>
  <entity class-name="acme.Employee" name="Employee" access="FIELD">
  <attributes>
  <id name="empid" />
  <basic name="name" />
  <basic name="salary" />
  <many-to-one name="department"
  target-entity="acme.Department"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
  </attributes>
  </entity>
  <entity class-name="acme.Department" name="Department" access="FIELD">
  <attributes>
  <id name="deptid" />
```

```
<basic name="name" />
<basic name="budget" />
<basic name="isManager" />
<one-to-many name="employees"
target-entity="acme.Employee"
fetch="LAZY" mapped-by="parentNode">
<cascade><cascade-persist/></cascade>
</one-to-many>
</attributes>
</entity>
</entity-mappings>
```

Defining indexes for non-entities using XML

Indexes for non-entity types are defined in XML. There is no difference when creating the MapIndexPlugin for entity maps and non-entity maps.

Java bean

```
public class Employee {
  int empid;
  String name
  double salary
  Department dept;

 public class Department {
  int deptid;
  String name;
  double budget;
  boolean isManager;
  Collection employees;
  }
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="DepartmentGrid">
  <backingMap name="Employee" pluginCollectionRef="Emp"/>
  <backingMap name="Department" pluginCollectionRef="Dept"/>
  <querySchema>
  <mapSchemas>
  <mapSchema mapName="Employee" valueClass="acme.Employee"
  primaryKeyField="empid" />
  <mapSchema mapName="Department" valueClass="acme.Department"
  primaryKeyField="deptid" />
  </mapSchemas>
  <relationships>
  <relationship source="acme.Employee"
  target="acme.Department"
  relationField="dept" invRelationField="employees" />
  </relationships>
  </querySchema>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Emp">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Employee.salary"/>
  <property name="AttributeName" type="java.lang.String" value="salary"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="Dept">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.name"/>
  <property name="AttributeName" type="java.lang.String" value="name"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Department.budget"/>
  <property name="AttributeName" type="java.lang.String" value="budget"/>
  <property name="RangeIndex" type="boolean" value="true"
  description="Ranges are must be set to true for attributes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Indexing relationships

WebSphere eXtreme Scale stores the foreign keys for related entities within the parent object. For entities, the keys are stored in the underlying tuple. For non-entity objects, the keys are explicitly stored in the parent object.

Adding an index on a relationship attribute can speed up queries that use cyclical references or use the IS NULL, IS EMPTY, SIZE and MEMBER OF query filters. Both single- and multi-valued associations may have the @Index annotation or a HashIndex plug-in configuration in an ObjectGrid descriptor XML file.

Defining entity relationship indexes using @Index

The following example defines entities with @Index annotations:

Entity with annotation

```
@Entity
public class Node {
    @ManyToOne @Index
    Node parentNode;

    @OneToMany @Index
    List<Node> childrenNodes = new ArrayList();

    @OneToMany @Index
    List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

Defining entity relationship indexes using XML

The following example defines the same entities and indexes using XML with HashIndex plug-ins:

Entity without annotations

```
  public class Node {
  int nodeId;
  Node parentNode;
  List<Node> childrenNodes = new ArrayList();
  List<BusinessUnitType> businessUnitTypes = new ArrayList();
  }
```

ObjectGrid XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
  <objectGrid name="ObjectGrid_Entity" entityMetadataXMLFile="entity.xml>
  <backingMap name="Node" pluginCollectionRef="Node"/>
  <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
  </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
  <backingMapPluginCollection id="Node">
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="parentNode"/>
  <property name="AttributeName" type="java.lang.String" value="parentNode"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />  </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="businessUnitType"/>
  <property name="AttributeName" type="java.lang.String" value="businessUnitTypes"/>

<property name="RangeIndex" type="boolean" value="false"
```

```
  description="Ranges are not supported for association indexes." />
 </bean>
  <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="childrenNodes"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodes"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
  </bean>
  </backingMapPluginCollection>
  </backingMapPluginCollections>
  </objectGridConfig>
```

Entity XML

```
  <?xml version="1.0" encoding="UTF-8"?>
  <entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

  <description>My entities</description>
  <entity class-name="acme.Node" name="Account" access="FIELD">
  <attributes>
  <id name="nodeId" />
  <one-to-many name="childrenNodes"
  target-entity="acme.Node"
  fetch="EAGER" mapped-by="parentNode">
  <cascade><cascade-all/></cascade>
  </one-to-many>
  <many-to-one name="parentNodes"
  target-entity="acme.Node"
  fetch="LAZY" mapped-by="childrenNodes">
  <cascade><cascade-none/></cascade>
  </one-to-many>
  <many-to-one name="businessUnitTypes"
  target-entity="acme.BusinessUnitType"
  fetch="EAGER">
  <cascade><cascade-persist/></cascade>
  </many-to-one>
</attributes>
  </entity>
  <entity class-name="acme.BusinessUnitType" name="BusinessUnitType" access="FIELD">
  <attributes>
  <id name="buId" />
  <basic name="TypeDescription" />
  </attributes>
  </entity>
  </entity-mappings>
```

Using the previously defined indexes, the following entity query examples are optimized:

```
SELECT n FROM Node n WHERE n.parentNode is null
SELECT n FROM Node n WHERE n.businessUnitTypes is EMPTY
  SELECT n FROM Node n WHERE size(n.businessUnitTypes)>=10
  SELECT n FROM BusinessUnitType b, Node n WHERE b member of n.businessUnitTypes and b.name='TELECOM'
```

Defining non-entity relationship indexes

The following example defines a HashIndex plug-in for non-entity maps in an ObjectGrid descriptor XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="ObjectGrid_POJO">
      <backingMap name="Node" pluginCollectionRef="Node"/>
      <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
      <querySchema>
        <mapSchemas>
          <mapSchema mapName="Node"
     valueClass="com.ibm.websphere.objectgrid.samples.entity.Node"
            primaryKeyField="id" />
          <mapSchema mapName="BusinessUnitType"
            valueClass="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            primaryKeyField="id" />
        </mapSchemas>
        <relationships>
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.Node"
            relationField="parentNodeId" invRelationField="childrenNodeIds" />
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            relationField="businessUnitTypeKeys" invRelationField="" />
        </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="Node">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNode"/>
<property name="Name" type="java.lang.String" value="parentNodeId"/>
<property name="AttributeName" type="java.lang.String" value="parentNodeId"/>
<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
      </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="businessUnitType"/>
        <property name="AttributeName" type="java.lang.String" value="businessUnitTypeKeys"/>

<property name="RangeIndex" type="boolean" value="false"
  description="Ranges are not supported for association indexes." />
   </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="childrenNodeIds"/>
  <property name="AttributeName" type="java.lang.String" value="childrenNodeIds"/>
  <property name="RangeIndex" type="boolean" value="false"
      description="Ranges are not supported for association indexes." />
</bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Given the above index configurations, the following object query examples are optimized:

```
SELECT n FROM Node n WHERE n.parentNodeId is null
SELECT n FROM Node n WHERE n.businessUnitTypeKeys is EMPTY
SELECT n FROM Node n WHERE size(n.businessUnitTypeKeys)>=10
SELECT n FROM BusinessUnitType b, Node n WHERE
 b member of n.businessUnitTypeKeys and b.name='TELECOM'
```

Indexing

Use the MapIndexPlugin to build an index or several indexes on a BackingMap to support non-key data access.

Index types and configuration

The indexing feature is represented by the MapIndexPlugin or Index for short. The Index is a BackingMap plug-in. A BackingMap can have multiple Index plug-ins configured, as long as each one follows the Index configuration rules.

You can use the indexing feature to build an index or several indexes on a BackingMap. An index is built from an attribute or a list of attributes of an object in the BackingMap. This feature provides a way for applications to find certain objects more quickly. With the indexing feature, applications can find objects with a specific value or within a range of values of indexed attributes.

Two types of indexing are possible: static and dynamic. With static indexing, you must configure the index plug-in on the BackingMap before initializing the ObjectGrid instance. You can do this configuration with XML or programmatic configuration of the BackingMap. Static indexing starts building an index during ObjectGrid initialization. The index is always synchronized with the BackingMap and ready for use. After the static indexing process starts, the maintenance of the index is part of the eXtreme Scale transaction management process. When transactions commit changes, these changes also update the static index, and index changes are rolled back if the transaction is rolled back.

With dynamic indexing, you can create an index on a BackingMap before or after the initialization of the containing ObjectGrid instance. Applications have life cycle control over the dynamic indexing process so that you can remove a dynamic index when it is no longer needed. When an application creates a dynamic index, the index might not be ready for immediate use because of the time it takes to complete the index building process. Because the amount of time depends upon the amount of data indexed, the DynamicIndexCallback interface is provided for applications that want to receive notifications when certain indexing events occur. These events include ready, error, and destroy. Applications can implement this callback interface and register with the dynamic indexing process.

If a BackingMap has an index plug-in configured, you can obtain the application index proxy object from the corresponding ObjectMap. Calling the getIndex method on the ObjectMap and passing in the name of the index plug-in returns the index proxy object. You must cast the index proxy object to an appropriate application index interface, such as MapIndex, MapRangeIndex, or a customized index interface. After obtaining the index proxy object, you can use methods defined in the application index interface to find cached objects.

The steps to use indexing are summarized in the following list:
- Add either static or dynamic index plug-ins into the BackingMap.
- Obtain an application index proxy object by issuing the getIndex method of the ObjectMap.
- Cast the index proxy object to an appropriate application index interface, such as MapIndex, MapRangeIndex, or a customized index interface.
- Use methods that are defined in application index interface to find cached objects.

For more information about writing your own index plug-in, see the information about writing an index plug-in in the *Programming Guide*..

For information about how to use indexing, see the information about using indexing for non-key data access in the *Programming Guide* and “Composite HashIndex” on page 123.

Data quality consideration

The results of index query methods only represent a snapshot of data at a point of time. No locks against data entries are obtained after the results return to the application. Application has to be aware that data updates may occur on a returned data set. For example, the application obtains the key of a cached object by running the findAll method of MapIndex. This returned key object is associated with a data entry in the cache. The application should be able to run the get method on ObjectMap to find an object by providing the key object. If another transaction removes the data object from the cache just before the get method is called, the returned result will be null.

Indexing performance considerations

One of the main objectives of the indexing feature is to improve overall BackingMap performance. If indexing is not used properly, the performance of the application might be compromised. Consider the following factors before using this feature.
- **The number of concurrent write transactions:** Index processing can occur every time a transaction writes data into a BackingMap. Performance degrades if many transactions are writing data into the map concurrently when an application attempts index query operations.

- **The size of the result set that is returned by a query operation:** As the size of the resultset increases, the query performance declines. Performance tends to degrade when the size of the result set is 15% or more of the BackingMap.
- **The number of indexes built over the same BackingMap:** Each index consumes system resources. As the number of the indexes built over the BackingMap increases, performance decreases.

The indexing function can improve BackingMap performance drastically. Ideal cases are when the BackingMap has mostly read operations, the query result set is of a small percentage of the BackingMap entries, and only few indexes are built over the BackingMap.

Using indexing for non-key data access

Using indexing as an alternative to key access for data can be more efficient.

Necessary steps

1. Add either static or dynamic index plug-ins to the BackingMap.
2. Obtain application index proxy object by issuing the getIndex method of ObjectMap.
3. Cast the index proxy object to an appropriate application index interface, such as MapIndex, MapRangeIndex, or a customized index interface.
4. Use the methods defined in the application index interface to find cached objects.

The HashIndex class is the index plug-in implementation that can support both of the built-in application index interfaces: MapIndex and MapRangeIndex. You can also create your own indexes.

Note: In a distributed environment, if the index object is obtained from a client ObjectGrid, it will have type client index object and all index operations will run in a remote server ObjectGrid. If the map is partitioned, the index operation will run on each partition remotely and results from each partition will be merged before returning them to the application. The performance will be determined by the number of partitions and the size of the result returned by each partition. Poor performance may occur if both factors are high.

If you want to write your own index plug-in, see “Writing an index plug-in” on page 146.

For information regarding indexing, see “Indexing” on page 118 and “Composite HashIndex” on page 123.

Adding static index plug-ins

You can use two approaches to add static index plug-ins into the BackingMap configuration: XML configuration and programmatic configuration. The following example illustrates the XML configuration approach.

Adding static index plug-ins: XML configuration approach

```
<backingMapPluginCollection id="person">
     <bean id="MapIndexplugin"
   className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="CODE"
      description="index name" />
        <property name="RangeIndex" type="boolean" value="true"
      description="true for MapRangeIndex" />
```

```
        <property name="AttributeName" type="java.lang.String" value="employeeCode"
     description="attribute name" />
      </bean>
</backingMapPluginCollection>
```

In this XML configuration example, the built-in HashIndex class is used as the index plug-in. The HashIndex supports properties that users can configure, such as Name, RangeIndex, and AttributeName in the previous example.

- The Name property is configured as "CODE", a string identifying this index plug-in. The Name property value must be unique within the scope of the BackingMap, and can be used to retrieve the index object by name from the ObjectMap instance for the BackingMap.
- The RangeIndex property is configured as "true", which means the application can cast the retrieved index object to the MapRangeIndex interface. If the RangeIndex property is configured as "false", the application can only cast the retrieved index object to the MapIndex interface. A MapRangeIndex supports functions to find data using range functions such as greater than, less than, or both, while a MapIndex only supports equals functions. If the index will be used by query, the RangeIndex property must be configured to "true" on single-attribute indexes. For a relationship index and composite index, the RangeIndex property must be configured to "false".
- The AttributeName property is configured as "employeeCode", which means the employeeCode attribute of the cached object is used to build a single-attribute index. If an application needs to search for cached objects with multiple attributes, the AttributeName property can be set to a comma-delimited list of attributes, yielding a composite index.

See the information about configuring HashIndex in the *Administration Guide* for more information.

The BackingMap interface has two methods that you can use to add static index plug-ins: addMapIndexplugin and setMapIndexplugins. For more information, see the API documentation.

The following code example illustrates the programmatic configuration approach:

Adding static index plugins: programmatic configuration approach

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

    ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
    ObjectGrid ivObjectGrid = ogManager.createObjectGrid( "grid" );
    BackingMap personBackingMap = ivObjectGrid.getMap("person");

    // use the builtin HashIndex class as the index plugin class.
    HashIndex mapIndexplugin = new HashIndex();
    mapIndexplugin.setName("CODE");
    mapIndexplugin.setAttributeName("EmployeeCode");
    mapIndexplugin.setRangeIndex(true);
    personBackingMap.addMapIndexplugin(mapIndexplugin);
```

Using static indexes

After a static index plug-in is added to a BackingMap configuration and the containing ObjectGrid instance is initialized, applications can retrieve the index object by name from the ObjectMap instance for the BackingMap. Cast the index object to the application index interface. Operations that the application index interface supports can now run.

The following code example illustrates how to retrieve and use static indexes.

Using static indexes example

```
Session session = ivObjectGrid.getSession();
    ObjectMap map = session.getMap("person ");
    MapRangeIndex codeIndex = (MapRangeIndex) m.getIndex("CODE");
    Iterator iter = codeIndex.findLessEqual(new Integer(15));
    while (iter.hasNext()) {
        Object key = iter.next();
        Object value = map.get(key);
    }
```

Adding, removing, and using dynamic indexes

You can create and remove dynamic indexes from a BackingMap instance programmatically at any time. A dynamic index differs from a static index in that the dynamic index can be created even after the containing ObjectGrid instance is initialized. Unlike static indexing, the dynamic indexing is an asynchronous process and needs to be in ready state before you use it. This method uses the same approach for retrieving and using the dynamic indexes as static indexes. You can remove a dynamic index if it is no longer needed. The BackingMap interface has methods to create and remove dynamic indexes.

See the BackingMap API for more information about the createDynamicIndex and removeDynamicIndex methods.

Using dynamic indexes example

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

        ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
        ObjectGrid og = ogManager.createObjectGrid("grid");
        BackingMap bm = og.getMap("person");
        og.initialize();

        // create index after ObjectGrid initialization without DynamicIndexCallback.
        bm.createDynamicIndex("CODE", true, "employeeCode", null);

        try {
            // If not using DynamicIndexCallback, need to wait for the Index to be ready.
            // The waiting time depends on the current size of the map
            Thread.sleep(3000);
        } catch (Throwable t) {
            // ...
        }

        // When the index is ready, applications can try to get application index
        // interface instance.
        // Applications have to find a way to ensure that the index is ready to use,
        // if not using DynamicIndexCallback interface.
        // The following example demonstrates the way to wait for the index to be ready
        // Consider the size of the map in the total waiting time.

        Session session = og.getSession();
        ObjectMap m = session.getMap("person");
        MapRangeIndex codeIndex = null;

        int counter = 0;
        int maxCounter = 10;
        boolean ready = false;
        while (!ready && counter < maxCounter) {
            try {
                counter++;
                codeIndex = (MapRangeIndex) m.getIndex("CODE");
                ready = true;
            } catch (IndexNotReadyException e) {
                // implies index is not ready, ...
                System.out.println("Index is not ready. continue to wait.");
                try {
                    Thread.sleep(3000);
                } catch (Throwable tt) {
                    // ...
                }
            } catch (Throwable t) {
                // unexpected exception
                t.printStackTrace();
            }
        }

        if (!ready) {
            System.out.println("Index is not ready.  Need to handle this situation.");
        }

        // Use the index to peform queries
        // Refer to the MapIndex or MapRangeIndex interface for supported operations.
        // The object attribute on which the index is created is the EmployeeCode.
        // Assume that the EmployeeCode attribute is Integer type: the
        // parameter that is passed into index operations has this data type.

        Iterator iter = codeIndex.findLessEqual(new Integer(15));

        // remove the dynamic index when no longer needed

        bm.removeDynamicIndex("CODE");
```

DynamicIndexCallback interface

The DynamicIndexCallback interface is designed for applications that want to get notifications at the indexing events of ready, error, or destroy. The DynamicIndexCallback is an optional parameter for the createDynamicIndex method of the BackingMap. With a registered DynamicIndexCallback instance, applications can run business logic upon receiving notification of an indexing event. For example, the ready event means that the index is ready for use. When a notification for this event is received, an application can try to retrieve and use the application index interface instance. See the DynamicIndexCallback API in the API documentation for more information.

The following code example illustrates the use of the DynamicIndexCallback interface:

Using DynamicIndexCallback interface

```
BackingMap personBackingMap = ivObjectGrid.getMap("person");
    DynamicIndexCallback callback = new DynamicIndexCallbackImpl();
    personBackingMap.createDynamicIndex("CODE", true, "employeeCode", callback);


    class DynamicIndexCallbackImpl implements DynamicIndexCallback {
        public DynamicIndexCallbackImpl() {
        }

        public void ready(String indexName) {
            System.out.println("DynamicIndexCallbackImpl.ready() -> indexName = " + indexName);

            // Simulate what an application would do when notified that the index is ready.
            // Normally, the application would wait until the ready state is reached and then proceed
            // with any index usage logic.
            if("CODE".equals(indexName)) {
                ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
                ObjectGrid og = ogManager.createObjectGrid( "grid" );
                Session session = og.getSession();
                ObjectMap map = session.getMap("person");
                MapIndex codeIndex = (MapIndex) map.getIndex("CODE");
                Iterator iter = codeIndex.findAll(codeValue);
            }
        }

        public void error(String indexName, Throwable t) {
            System.out.println("DynamicIndexCallbackImpl.error() -> indexName = " + indexName);
            t.printStackTrace();
        }

        public void destroy(String indexName) {
            System.out.println("DynamicIndexCallbackImpl.destroy() -> indexName = " + indexName);
        }
    }
```

Composite HashIndex

The composite HashIndex improves query performance and avoids expensive map scanning. The feature also provides a convenient way for the HashIndex API to find cached objects when search criteria involve many attributes.

Improved performance

A composite HashIndex provides a fast and convenient way to search for cached objects with multiple attributes in match-searching criteria. The composite index supports full attribute-match searches, but does not support range searches.

Note: Composite indexes do not support the BETWEEN operator in the ObjectGrid query language because BETWEEN would require range support. The greater than (>) and less than (<) conditionals also do not work because they require range indexes.

A composite index can improve performance of queries if the appropriate composite index is available for the WHERE condition. This means that the composite index has exactly the same attributes as involved in the WHERE condition with full attributes matched.

A query might have many attributes involved in a condition as in the following example.

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'
```

Composite index can improve query performance by avoiding scanning map or joining multiple single-attribute index results. In the example, if a composite index is defined with attributes (city,state,zipcode), the query engine can use the composite index to find the entry with `city='Rochester'`, `state='MN'`, and `zipcode='55901'`. Without composite index and attribute index on city, state, and zipcode attributes, the query engine will have to scan the map or join multiple single-attribute searches, which usually have expensive overhead. Also, querying for the composite index only supports a full-matched pattern.

Configuring a composite index

You can configure composite indexing in three ways: using XML, programmatically, and (for entity maps only) with entity annotations.

Using XML

In order to configure a composite index with XML, include code such as below in the configuration file's backingMapPluginCollections element.

Composite index - XML configuration approach

```
<bean id="MapIndexPlugin"  className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Address.CityStateZip"/>
<property name="AttributeName" type="java.lang.String" value="city,state,zipcode"/>
</bean>
```

Programmatic configuration

The programmatic example code below will create the same composite index as the preceding XML.

```
   HashIndex mapIndex = new HashIndex();
    mapIndex.setName("Address.CityStateZip");
    mapIndex.setAttributeName(("city,state,zipcode"));
    mapIndex.setRangeIndex(true);

    BackingMap bm = objectGrid.defineMap("mymap");
    bm.addMapIndexPlugin(mapIndex);
```

Note that configuring a composite index is the same as configuring a regular index with XML except for the attributeName property value. In a composite index case, the value of attributeName is a comma-delimited list of attributes. For example, the value class Address has 3 attributes: city, state, and zipcode. A composite index can be defined with the attributeName property value as `"city,state,zipcode"` indicating that city, state, and zipcode are included in the composite index.

Also, note that the composite HashIndexes do not support range lookups and therefore cannot have the RangeIndex property set to true.

With entity annotations

In the entity map case, annotation approach can be used to define a composite index. You can define a list of CompositeIndex within CompositeIndexes annotation on the entity class level. The CompositeIndex has a name and attributeNames property. Each CompositeIndex is associated with a HashIndex instance applied to the entity's associated BackingMap. The HashIndex is configured as a non-range index.

```
@Entity
@CompositeIndexes({
    @CompositeIndex(name="CityStateZip", attributeNames="city,state,zipcode"),
    @CompositeIndex(name="lastnameBirthday", attributeNames="lastname,birthday")
 })
public class Address {
    @Id int id;
    String street;
    String city;
    String state;
    String zipcode;
    String lastname;
    Date birthday;
}
```

The name property for each composite index must be unique within the entity and BackingMap. If the name is not specified, a generated name will be used. The attributeNames property is used to populate the HashIndex attributeName with the comma-delimited list of attributes. The attribute names coincide with the persistent field names when the entities are configured to use field-access, or the property name as defined for the JavaBeans naming conventions for property-access entities. For example: If the attribute name is "street", the property getter method is named getStreet.

Performing composite index lookups

After a composite index is configured, an application can use the findAll(Object) method of the MapIndex interface to perform lookups, as below.

```
Session sess = objectgrid.getSession();
ObjectMap map = sess.getMap("MAP_NAME");
MapIndex codeIndex = (MapIndex) map.getIndex("INDEX_NAME");
Object[] compositeValue = new Object[]{ MapIndex.EMPTY_VALUE,
   "MN", "55901"};
Iterator iter = mapIndex.findAll(compositeValue);
```

The MapIndex.EMPTY_VALUE is assigned to the compositeValue[0] which indicates that the city attribute is excluded from evaluation. Only objects with state attribute equal to "MN" and zipcode attribute equal to "55901" will be included in the result.

The following queries benefit from the previous composite index configuration:

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'

SELECT a FROM Address a WHERE a.state='MN' AND a.zipcode='55901'
```

The query engine will find the appropriate composite index and use it to improve query performance in full attribute-match cases.

In some scenarios, the application might need to define multiple composite indexes with overlapped attributes in order to satisfy all queries with full attributes matched. A disadvantage of increasing the number of indexes is the possible performance overhead on map operations.

Migration and interoperability

The only constraint for the use of a composite index is that an application cannot configure it in a distributed environment with heterogeneous containers. Old and new containers cannot be mixed, since older containers will not recognize a composite index configuration. The composite index is just like the existing regular attribute index, except that the former allows indexing over multiple attributes. When using only the regular attribute index, a mixed-container environment is still viable.

Data Grid API

The DataGrid API provides a simple programming interface to run business logic over all or a subset of the ObjectGrid in parallel with where the data is located.

DataGrid APIs and partitioning

With the DataGrid APIs, a client can send requests to one partition, a subset of partitions, or all the partitions in a data grid. The client can specify a list of keys, and WebSphere eXtreme Scale determines the set of partitions that are hosting the keys. The request is then sent to all the partitions in the set in parallel and the client waits for the results. The client can also send requests without specifying keys, therefore, requests are sent to all partitions.

Agents that are deployed to the data grid do not work in client mode. These agents work directly against the primary shard. Working directly against the primary shard results in maximum performance, allowing tens of thousands or more transactions per second because the agent works with the data at full memory speeds. Working directly with the primary shard also means that an agent can only see data that is within that shard. This provides some interesting opportunities that cannot be done on a client.

A typical eXtreme Scale client must be able determine the partition from the transaction, because the client needs to route the request. If an agent is directly attached to a shard, then no routing is needed. All requests go against that shard. Because the agent is directly attached to a shard, data in other maps in the shard can be accessed without worrying about common partitioning keys, and so on, because no routing occurs.

DataGrid agents and entity-based Maps

A map contains key objects and value objects. The key object is a generated tuple as is the value. An agent is normally provided with the application specified key objects.

The key object is a generated tuple as is the value. An agent is normally provided with the application specified key objects. This will be the key objects used by the application or Tuples if it is an entity Map. An application using Entities will not want to deal with Tuples directly and would prefer to work with the Java objects mapped to the Entity.

Therefore, an Agent class can implement the EntityAgentMixin interface. This forces the class to implement one more method, getClassForEntity(). This returns the entity class to use with the agent on the server side. The keys are converted to this Entity before invoking the process and reduce methods.

This is a different semantic to an non EntityAgentMixin agent where those methods are provided with just the keys. An agent implementing EntityAgentMixin receives the Entity object which includes keys and values in one object.

Note: If the entity does not exist on the server, the keys are the raw Tuple format of the key instead of the managed entity.

DataGrid API example

The DataGrid APIs support two common grid programming patterns: parallel map and parallel reduce.

Parallel Map

The parallel map allows the entries for a set of keys to be processed and returns a result for each entry processed. The application makes a list of keys and receives a Map of key/result pairs after invoking a Map operation. The result is the result of applying a function to the entry of each key. The function is supplied by the application.

MapGridAgent call flow

When the AgentManager.callMapAgent method is invoked with a collection of keys, the MapGridAgent instance is serialized and sent to each primary partition that the keys resolve to. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The process method is invoked for each instance one time for each key that resolves to the partition. The result of each process method is then serialized back to the client and returned to the caller in a Map instance, where the result is represented as the value in the map.

When the AgentManager.callMapAgent method is invoked without a collection of keys, the MapGridAgent instance is serialized and sent to every primary partition. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance (partition) of the agent. The processAllEntries method is invoked for each partition. The result of each processAllEntries method is then serialized back to the client and returned to the caller in a Map instance. The following example assumes there is a Person entity with the following shape:

```
import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
@Entity
public class Person
{
  @Id String ssn;
  String firstName;
  String surname;
  int age;
}
```

The application supplied function is written as a class implementing the MapAgentGrid interface. Following is an example agent showing a function to return the age of a Person multiplied by two.

```
public class DoublePersonAgeAgent implements MapGridAgent, EntityAgentMixin
{
  private static final long serialVersionUID = -2006093916067992974L;

  int lowAge;
  int highAge;

  public Object process(Session s, ObjectMap map, Object key)
  {
    Person p = (Person)key;
    return new Integer(p.age * 2);
  }

  public Map processAllEntries(Session s, ObjectMap map)
  {
    EntityManager em = s.getEntityManager();
    Query q = em.createQuery("select p from Person p where p.age > ?1 and p.age < ?2");
    q.setParameter(1, lowAge);
    q.setParameter(2, highAge);
    Iterator iter = q.getResultIterator();
    Map<Person, Interger> rc = new HashMap<Person, Integer>();
   while(iter.hasNext())
    {
   Person p = (Person)iter.next();
   rc.put(p, (Integer)process(s, map, p));
  }
  return rc;
 }
 public Class getClassForEntity()
 {
  return Person.class;
 }
}
```

This shows the Map agent for doubling a Person. Lets look at the process methods first. The first process method is supplied with the Person to work with. It simply returns double the age of that entry. The second process method is called for each partition and finds all Person objects with an age between lowAge and highAge and returns their ages doubled.

```
Session s = grid.getSession();
ObjectMap map = s.getMap("Person");
AgentManager amgr = map.getAgentManager();

DoublePersonAgeAgent agent = new DoublePersonAgeAgent();

// make a list of keys
ArrayList<Person> keyList = new ArrayList<Person>();
Person p = new Person();
p.ssn = "1";
keyList.add(p);
p = new Person ();
p.ssn = "2";
keyList.add(p);

// get the results for those entries
Map<Tuple, Object> = amgr.callMapAgent(agent, keyList);
```

This shows a client obtaining a Session and a reference to the Person Map. The agent operation is performed against a specific Map. The AgentManager interface is retrieved from that Map. An instance of the agent to invoke is created and any necessary state is added to the object by setting attributes, there are none in this case. A list of keys are then constructed. A Map with the values for person 1 doubled, and the same values for person 2 are returned.

The agent is then invoked for that set of keys. The agents process method is invoked on each partition with some of the specified keys in the grid in parallel. A

Map is returned providing the merged results for the specified key. In this case, a Map with the values holding the age for person 1 doubled and the same for person 2 will be returned.

If the key does not exist, the agent will still be invoked. This gives the agent the opportunity to create the map entry. If using an EntityAgentMixin, the key to process will not be the entity, but will instead be the actual Tuple key value for the entity. If the keys are unknown then it's possible to ask all partitions to find Person objects of a certain shape and return their ages doubled. Here is an example:

```
Session s = grid.getSession();
  ObjectMap map = s.getMap("Person");
  AgentManager amgr = map.getAgentManager();

  DoublePersonAgeAgent agent = new DoublePersonAgeAgent();
  agent.lowAge = 20;
  agent.highAge = 9999;

  Map m = amgr.callMapAgent(agent);
```

The previous example shows the AgentManager being obtained for the Person Map, and the agent constructed and initialized with the low and high ages for Persons of interest. The agent is then invoked using the callMapAgent method. Notice, no keys are supplied. This causes the ObjectGrid to invoke the agent on every partition in the grid in parallel and then return the merged results to the client. This will find all Person objects in the grid with an age between low and high and calculate the age of those Person objects doubled. This shows how the grid apis can be used to run a query to find entities matching a certain query. The agent is simply serialized and transported by the ObjectGrid to the partitions with the needed entries. The results are similarly serialized for transport back to the client. Care needs to be taken with the Map APIs. If the ObjectGrid was hosting tera bytes of objects and running on a lot of servers then potentially this would overwhelm anything but the largest machines running the client. This should be used to processing a small subset. If a large subset needs processing then we recommend using a reduce agent to do the processing out in the grid rather than on a client.

Parallel Reduction or aggregation agents

This style of programming processes a subset of the entries and calculates a single result for the group of entries. Examples of such a result would be:

- minimum value
- maximum value
- some other business specific function

A reduce agent is coded and invoked in a very similar manner to the Map agents.

ReduceGridAgent call flow

When the AgentManager.callReduceAgent method is invoked with a collection of keys, the ReduceGridAgent instance is serialized and sent to each primary partition that the keys resolve to. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The reduce(Session s, ObjectMap map, Collection keys) method is invoked once per instance (partition) with the subset of keys that resolves to the partition. The result of each reduce method is then serialized back to the client. The reduceResults method is invoked on the client ReduceGridAgent instance with the collection of each result from each remote reduce invocation. The result from the reduceResults method is returned to the caller of the callReduceAgent method.

When the AgentManager.callReduceAgent method is invoked without a collection of keys, the ReduceGridAgentinstance is serialized and sent to each primary partition. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The reduce(Session s, ObjectMap map) method is invoked once per instance (partition). The result of each reduce method is then serialized back to the client. The reduceResults method is invoked on the client ReduceGridAgent instance with the collection of each result from each remote reduce invocation. The result from the reduceResults method is returned to the caller of the callReduceAgent method. Here is an example of a reduce agent that simply adds the ages of the matching entries.

```
public class SumAgeReduceAgent implements ReduceGridAgent, EntityAgentMixin
{
  private static final long serialVersionUID = 2521080771723284899L;

  int lowAge;
  int highAge;

  public Object reduce(Session s, ObjectMap map, Collection keyList)
  {
    Iterator<Person> iter = keyList.iterator();
  int sum = 0;
  while (iter.hasNext())
  {
   Person p = iter.next();
   sume += p.age;
  }
  return new Integer(sum);
 }

 public Object reduce(Session s, ObjectMap map)
 {
  EntityManager em = s.getEntityManager ();
  Query q = em.createQuery("select p from Person p where p.age > ?1 and p.age < ?2");
  q.setParameter(1, lowAge);
  q.setParameter(2, highAge);
  Iterator<Person> iter = q.getResultIterator();
  int sum = 0;
  while(iter.hasNext())
  {
   sum += iter.next().age;
  }
  return new Integer(sum);
 }

 public Class getClassForEntity()
 {
  return Person.class;
 }
}
```

The previous example shows the agent. The agent has three important parts. The first allows a specific set of entries to be processed without a query. It simply interates over the set of entries adding the ages. The sum is returned from the method. The second uses a query to select the entries to be aggregated. It then sums all the matching Person ages. The third method is used to aggregate the results from each partition to a single result. The ObjectGrid performs the entry aggregation in parallel across the grid. Each partition produces an intermediate result that must be aggregated with other partition intermediate results. This third method performs that task. In the following example the agent is invoked, and the ages of all Persons with ages between 10 and 20 exclusively are aggregrated:

```
Session s = grid.getSession();
  ObjectMap map = s.getMap("Person");
  AgentManager amgr = map.getAgentManager();

  SumAgeReduceAgent agent = new SumAgeReduceAgent();

  Person p = new Person();
  p.ssn = "1";
```

```
 ArrayList<Person> list = new ArrayList<Person>();
list.add(p);
p = new Person ();
p.ssn = "2";
list.add(p);
Integer v = (Integer)amgr.callReduceAgent(agent, list);
```

Agent functions

The agent is free to do ObjectMap or EntityManager operations within the local shard where it is running. The agent receives a Session and can add, update, query, read, or remove data from the partition the Session represents. Some applications will only query data from the grid, but you can also write an agent to increment all the Person ages by 1 that match a certain query. There is a transaction on the Session when the agent is called, and is committed when the agent returns unless an exception is thrown

Error handling

If a map agent is invoked with an unknown key then the value that is returned is an error object implementing the EntryErrorValue interface.

Transactions

A map agent runs in a separate transaction from the client. Agent invocations may be grouped into a single transaction. If an agent fails (throws an exception), the transaction is rolled-back. Any agents that ran successfully in a transaction will rollback with the failed agent. The AgentManager will rerun the rolled-back agents that ran successfully in a new transaction.

For more information, consult the DataGrid API documentation.

API Documentation

The WebSphere eXtreme Scale API contains information you can use to look up a package or class name to find details about systems or application programming interfaces.

See the WebSphere eXtreme Scale Information Center for the API documentation.

Chapter 4. System APIs and plug-ins

A plug-in is a component that provides a function to the pluggable components, which include ObjectGrid and BackingMap. To most effectively use eXtreme Scale as an in-memory data grid or database processing space, you should carefully determine how best you can maximize performance with available plug-ins.

Introduction to plug-ins

A WebSphere eXtreme Scale plug-in is a component that provides a certain type of function to the pluggable components that include ObjectGrid and BackingMap. WebSphere eXtreme Scale provides several plug points to allow applications and cache providers to integrate with various data stores, alternative client APIs and to improve overall performance of the cache. The product ships with several default, prebuilt plug-ins, but you can also build custom plug-ins with the application.

All plug-ins are concrete classes that implement one or more eXtreme Scale plug-in interfaces. These classes are then instantiated and invoked by the ObjectGrid at appropriate times. The ObjectGrid and BackingMaps each allow custom plug-ins to be registered.

ObjectGrid plug-ins

The following plug-ins are available for an ObjectGrid instance:

- **TransactionCallback**: A TransactionCallback plug-in provides transaction life cycle events.
- **ObjectGridEventListener**: An ObjectGridEventListener plug-in provides ObjectGrid life cycle events for the ObjectGrid, shards, and transactions.
- **SubjectSource, ObjectGridAuthorization, SubjectValidation**: eXtreme Scale provides several security endpoints to allow custom authentication mechanisms to be integrated with eXtreme Scale.

Common ObjectGrid plug-in requirements

The ObjectGrid instantiates and initializes plug-in instances using JavaBeans conventions. All of the previous plug-in implementations have the following requirements:

- The plug-in class must be a top-level public class.
- The plug-in class must provide a public, no-argument constructor.
- The plug-in class must be available in the class path for both servers and clients (as appropriate).
- Attributes must be set using the JavaBeans style property methods.
- Plug-ins, unless specifically noted, are registered before ObjectGrid initializes and cannot be changed after the ObjectGrid is initialized.

BackingMap plug-ins

The following plug-ins are available for a BackingMap:

- **Evictor**: An evictor plug-in is a default mechanism is provided for evicting cache entries and a plug-in for creating custom evictors.

- **Loader**: A Loader plug-in on an ObjectGrid map acts as a memory cache for data that is typically kept in a persistent store on either the same system or some other system.
- **ObjectTransformer**: An ObjectTransformer plug-in allows you to serialize, deserialize, and copy objects in the cache.
- **OptimisticCallback**: An OptimisticCallback plug-in allows you to customize versioning and comparison operations of cache objects when you are using the optimistic lock strategy.
- **MapEventListener**: A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap.
- **Indexing**: Use the indexing feature, which is represented by the MapIndexplug-in plug-in, to build an index or several indexes on a BackingMap map to support non-key data access.

Plug-in life cycles

Most plug-ins have both initialize and destroy methods or equivalent methods, in addition to the methods for which they were designed to operate. These specialized methods of each plug-in are available to be invoked at designated functional points. Both initialize and destroy methods define the life cycle of plug-ins, which are controlled by their "owner" objects. An owner object is the object that actually uses the given plug-in. An owner can be a grid client, server, or a backing map.

When owner objects are initializing, they will invoke the initialize method of their owned plug-ins. During the destroy cycle of owner objects, the destroy method of plug-ins will be consequently invoked also. For details on the specifics of initialize and destroy methods, along with other methods capable with each plug-in, refer to the topics relevant to each plug-in.

As an example, consider a distributed environment. Both the client-side ObjectGrids and the server-side ObjectGrids can have their own plug-ins. The life cycle of a client-side ObjectGrid and, therefore, its plug-in instances are independent from all server-side ObjectGrid and plug-in instances.

In such a distributed topology, say you have an ObjectGrid named "myGrid" defined in the objectGrid.xml file and configured with a customized ObjectGridEventListener named myObjectGridEventListener. The objectGridDeployment.xml file defines the deployment policy for the myGrid ObjectGrid. Both objectGrid.xml and objectGridDeployment.xml are used to start container servers. During the startup of the container server, the server side myGrid ObjectGrid instance will be initialized and the initialize method of the myObjectGridEventListener instance owned by the myObjectGrid instance will be invoked. After the container server is started, your application can connect to the server-side myGrid ObjectGrid instance and obtain a client-side instance.

When obtaining the client-side myGrid ObjectGrid instance, the client-side myGrid instance will go through its own initialization cycle and invoke the initialize method of its own client-side myObjectGridEventListener instance. This client side myObjectGridEventListener instance is independent from the server-side myObjectGridEventListener instance. Its life cycle is controlled by its owner, which is the client-side myGrid ObjectGrid instance.

If your application disconnects or destroys the client-side myGrid ObjectGrid instance, the destroy method of the owned client-side myObjectGridEventListener instance will be invoked automatically. However, this has no impact on server-side myObjectGridEventListener instance. The destroy method of the server-side myObjectGridEventListener instance will only be invoked during the destroy cycle of the server-side myGrid ObjectGrid instance when stopping a container server. That is, when stopping a container server, the contained ObjectGrid instances will be destroyed and the destroy method of all their owned plug-ins will be invoked.

Although the previous example applies specifically to the case of a client and a server instance of an ObjectGrid, the owner of a plug-in can also be a BackingMap and you must be careful to determine your configurations for plug-ins that you may write based on these life cycle considerations.

Event listeners

You can use the ObjectGridEventListener and MapEventListener plug-ins to configure notifications for various events in the eXtreme Scale cache. Listener plug-ins are registered with an ObjectGrid or BackingMap instance like other eXtreme Scale plug-ins and add integration and customization points for applications and cache providers.

ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in provides eXtreme Scale life cycle events for the ObjectGrid instance, shards, and transactions. Use the ObjectGridEventListener plug-in to receive notifications when significant events occur on an ObjectGrid. These events include ObjectGrid initialization, the beginning of a transaction, the ending a transaction, and destroying an ObjectGrid. To listen for these events, create a class that implements the ObjectGridEventListener interface and add it to the eXtreme Scale.

For more information about writing an ObjectGridEventListener plug-in, see "ObjectGridEventListener plug-in" on page 137. You can also refer to the API documentation for more information.

Adding and removing ObjectGridEventListener instances

An ObjectGrid can have multiple ObjectGridEventListener listeners. Add and remove the listeners using the addEventListener, setEventListeners and removeEventListener methods on the ObjectGrid interface. You can also declaratively registerObjectGridEventListener plug-ins with the ObjectGrid descriptor file. For examples, see "ObjectGridEventListener plug-in" on page 137.

MapEventListener plug-in

A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap instance. For details on writing a MapEventListener plug-in, see "MapEventListener plug-in" on page 136. You can also refer to the API documentation for more information.

Adding and removing MapEventListener instances

An eXtreme Scale can have multiple MapEventListener listeners. Add and remove listeners with the addMapEventListener, setMapEventListeners and removeMapEventListener methods on the BackingMap interface. You can also declaratively register MapEventListener listeners with the ObjectGrid descriptor file. For examples, see "MapEventListener plug-in" on page 136.

MapEventListener plug-in

A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap object: when a map has finished pre-loading or when an entry is evicted from the map. A particular MapEventListener plug-in is a custom class you write implementing the MapEventListener interface.

MapEventListener plug-in conventions

When you develop a MapEventListener plug-in, you must follow common plug-in conventions. For more information about plug-in conventions, see “Introduction to plug-ins” on page 133. For other types of listener plug-ins, see “Event listeners” on page 135.

After you write a MapEventListener implementation, you can plug it in to the BackingMap configuration programmatically or with an XML configuration.

Write a MapEventListener implementation

Your application can include an implementation of the MapEventListener plug-in. The plug-in must implement the MapEventListener interface to receive significant events about a map. Events are sent to the MapEventListener plug-in when an entry is evicted from the map and when the preload of a map completes.

Plug in a MapEventListener implementation using XML

A MapEventListner implementation can be configured using XML. The following XML must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <backingMap name="myMap" pluginCollectionRef="myPlugins" />
        </objectGrid>
    </objectGrids>
    <backingMapPluginCollections>
        <backingMapPluginCollection id="myPlugins">
            <bean id="MapEventListener" className=
       "com.company.org.MyMapEventListener" />
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

Providing this file to the ObjectGridManager instance facilitates the creation of this configuration. The following code snippet shows how to create an ObjectGrid instance using this XML file. The newly created ObjectGrid instance has a MapEventListener set on the myMap BackingMap.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid =
 objectGridManager.createObjectGrid("myGrid", new URL("file:etc/test/myGrid.xml"),
  true, false);
```

Programmatically plug in a MapEventListener implementation

The class name for the custom MapEventListener is the com.company.org.MyMapEventListener class. This class implements the

MapEventListener interface. The following code snippet creates the custom MapEventListener object and adds it to a BackingMap object:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap myMap = myGrid.defineMap("myMap");
MyMapEventListener myListener = new MyMapEventListener();
myMap.addMapEventListener(myListener);
```

ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in provides WebSphere eXtreme Scale life cycle events for the ObjectGrid, shards and transactions. An ObjectGridEventListener plug-in provides notifications when an ObjectGrid is initialized or destroyed, and when a transaction is started or ended. ObjectGridEventListener plug-ins are custom classes you write implementing the ObjectGridEventListener interface. Optionally, the implementation includes ObjectGridEventGroup sub-interfaces and follow the common eXtreme Scale plug-in conventions.

Overview

An ObjectGridEventListener plug-in is useful when a Loader plug-in is available, and you must initialize Java Database Connectivity (JDBC) connections or connections to a back end when transactions start and end. Typically, an ObjectGridEventListener plug-in and a Loader plug-in are written together.

Writing an ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in must implement the ObjectGridEventListener interface to receive notifications about significant eXtreme Scale events. To receive additional event notifications, you can implement the following interfaces. These sub-interfaces are included in the ObjectGridEventGroup interface:

- ShardEvents interface
- ShardLifecycle interface
- TransactionEvents interface

For more information about these interfaces, see the API documentation.

Shard events

When the catalog service places partition primary or replica shards in a Java virtual machine (JVM), a new ObjectGrid instance is created in that JVM to host that shard. Some applications that need to start threads on the JVM host the primary need notification of these events. The ObjectGridEventGroup.ShardEvents interface declares the shardActivate and shardDeactivate methods. These methods are called only when a shard is activated as a primary and when the shard is deactivated from a primary. These two events allow the application to start additional threads when the shard is a primary and stop the threads when the shard returns to being a replica or is taken out of service.

An application can determine which partition has been activated by looking up a specific BackingMap in the ObjectGrid reference that is provided to the shardActivate method using the ObjectGrid#getMap method. The application can then see the partition number using the BackingMap#getPartitionId() method. The partitions are numbered from 0 to the number of partitions in the deployment descriptor minus one.

Shard life-cycle events

ObjectGridEventListener.initialize and ObjectGridEventListener.destroy method events are delivered using the ObjectGridEventGroup.ShardLifecycle interface.

Transaction events

ObjectGridEventListener.transactionBegin and ObjectGridEventListener.transactionEnd methods are delivered through the ObjectGridEventGroup.TransactionEvents interface.

Advantages of this approach

If an ObjectGridEventListener plug-in implements the ObjectGridEventListener and ShardLifecycle interfaces, then shard life-cycle events are the only events that are delivered to the listener. After you implement any of the new ObjectGridEventGroup inner interfaces, eXtreme Scale only delivers those specific events by the new interfaces. With this implementation, code can be backwards compatible. If you are using the new inner interfaces, it can now receive just the specific events that are needed.

Using the ObjectGridEventListener plug-in

To use a custom ObjectGridEventListener plug-in, first create a class that implements the ObjectGridEventListener interface and any optional ObjectGridEventGroup sub-interfaces. Add the custom listener to an ObjectGrid to receive notification of significant events. You have two approaches to add an ObjectGridEventListener plug-in into the eXtreme Scale configuration: programmatic configuration and XML configuration.

Configure an ObjectGridEventListener plug-in programmatically

Assume that the class name of the eXtreme Scale event listener is the com.company.org.MyObjectGridEventListener class. This class implements the ObjectGridEventListener interface. The following code snippet creates the custom ObjectGridEventListener and adds it to an ObjectGrid.

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
MyObjectGridEventListener myListener = new MyObjectGridEventListener();
myGrid.addEventListener(myListener);
```

Configure an ObjectGridEventListener plug-in with XML

You can also configure an ObjectGridEventListener plug-in using XML. The following XML creates a configuration that is equivalent to the described programmatically created ObjectGrid event listener. The following text must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <bean id="ObjectGridEventListener"
       className="com.company.org.MyObjectGridEventListener" />
            <backingMap name="Book"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

Notice the bean declarations come before the backingMap declarations. Provide this file to the ObjectGridManager plug-in to facilitate the creation of this configuration. The following code snippet demonstrates how to create an ObjectGrid instance using this XML file. The ObjectGrid instance that is created has an ObjectGridEventListener listener set on the myGrid ObjectGrid.

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid",
 new URL("file:etc/test/myGrid.xml"), true, false);
```

Eviction

WebSphere eXtreme Scale provides a default mechanism for evicting cache entries and a plug-in for creating custom evictors. An evictor controls the membership of entries in each BackingMap. The default evictor uses a time to live (TTL) eviction policy for each BackingMap. If you provide a pluggable evictor mechanism, it typically uses an eviction policy that is based on the number of entries instead of on time.

TimeToLive property

A default TTL evictor is created with every backing map. The default evictor removes entries based on a time to live concept. This behavior is defined by the ttlType attribute, which has three types.

- None: Specifies that entries never expire and therefore are never removed from the map.
- Creation time: Specifies that entries are evicted depending on when they were created.
- Last accessed time: Specifies that entries are evicted depending upon when they were last accessed.

If you are using the CREATION_TIME ttlType, the evictor evicts an entry when its time from creation equals its TimeToLive attribute value (which is set in milliseconds in your application configuration). If you set the TimeToLive attribute value to 10 seconds, the entry is automatically evicted ten seconds after it was inserted. It is important to take caution when setting this value for the CREATION_TIME ttlType. This evictor is best used when reasonably high amounts of additions to the cache exist that are only used for a set amount of time. With this strategy, anything that is created is removed after the set amount of time.

Following is an example of where a TTL type of CREATION_TIME is useful. You are using a Web application that obtains stock quotes, and getting the most recent quotes is not critical. In this case, the stock quotes are cached in an ObjectGrid for 20 minutes. After 20 minutes, the ObjectGrid map entries expire and are evicted. Every twenty minutes or so the ObjectGrid map uses the Loader plug-in to refresh the map data with fresh data from the database. The database is updated every 20 minutes with the most recent stock quotes. So for this application, using a TimeToLive value of 20 minutes is ideal.

If you are using the LAST_ACCESSED_TIME ttlType attribute, set the TimeToLive to a lower number than if you are using the CREATION_TIME ttlType, because the entries TimeToLive attribute is reset every time it is accessed. In other words, if the TimeToLive attribute is equal to 15 and an entry has existed for 14 seconds but then gets accessed, it does not expire again for another 15 seconds. If you set the TimeToLive to a relatively high number, many entries might never be evicted. However, if you set the value to something like 15 seconds, entries might be removed when they are not often accessed.

Following is an example of where a TTL type of LAST_ACCESSED_TIME is useful. An ObjectGrid map is used to hold session data from a client. Session data must be destroyed if the client does not use the session data for some period of time. For example, the session data times out after 30 minutes of no activity by the client. In this case, using a TTL type of LAST_ACCESSED_TIME with the TimeToLive attribute set to 30 minutes is exactly what is needed for this application.

The following example creates a backing map, set its default evictor ttlType attribute, and sets its TimeToLive property.

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");
bMap.setTtlEvictorType(TTLType.LAST_ACCESSED_TIME);
bMap.setTimeToLive(1800);
```

Most evictor settings should be set before you initialize the ObjectGrid.

You may also write your own evictors: For more information, see the information about writing a custom evictor in the *Programming Guide*.

Optional evictors

The default TTL evictor uses an eviction policy that is based on time, and the number of entries in the BackingMap has no affect on the expiration time of an entry. You can use an optional pluggable evictor to evict entries based on the number of entries that exist instead of based on time.

The following optional pluggable evictors provide some commonly used algorithms for deciding which entries to evict when a BackingMap grows beyond some size limit. *
- The LRUEvictor evictor uses a least recently used (LRU) algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.
- The LFUEvictor evictor uses a least frequently used (LFU) algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.

The BackingMap informs an evictor as entries are created, modified, or removed in a transaction. The BackingMap keeps track of these entries and chooses when to evict one or more entries from the BackingMap.

A BackingMap has no configuration information for a maximum size. Instead, evictor properties are set to control the evictor behavior. Both the LRUEvictor and the LFUEvictor have a maximum size property that is used to cause the evictor to begin to evict entries after the maximum size is exceeded. Like the TTL evictor, the LRU and LFU evictors might not immediately evict an entry when the maximum number of entries is reached to minimize impact on performance.

If the LRU or LFU eviction algorithm is not adequate for a particular application, you can write your own evictors to create your eviction strategy.

Memory-based eviction

Important: Memory-based eviction is only supported on Java Platform, Enterprise Edition Version 5 or later.

All built-in evictors support memory-based eviction that can be enabled on the BackingMap interface by setting the evictionTriggers attribute of BackingMap to `MEMORY_USAGE_THRESHOLD`. For more information about how to set the evictionTriggers attribute on BackingMap, see the information about the BackingMap interface and the ObjectGrid descriptor XML file in the *Administration Guide*.

Memory-based eviction is based on heap usage threshold. When memory-based eviction is enabled on BackingMap and the BackingMap has any built-in evictor, the usage threshold is set to a default percentage of total memory if the threshold has not been previously set.

When you are using memory-based eviction, you should configure the garbage collection threshold to the same value as their target heap utilization. For example, if the memory-based eviction threshold is set at 50 percent and the garbage collection threshold is at the default 70 percent level, then the heap utilization can go as high as 70 percent. This heap utilization increase occurs because memory-based eviction is only triggered after a garbage collection cycle.

The memory-based eviction algorithm used by WebSphere eXtreme Scale is sensitive to the behavior of the garbage collection algorithm in use. The best algorithm for memory-based eviction is the IBM default throughput collector. Generation garbage collection algorithms can cause undesired behavior, and so you should not use these algorithms with memory-based eviction.

To change the usage threshold percentage, set the memoryThresholdPercentage property on the container and server property files for eXtreme Scale server processes.

During runtime, if the memory usage exceeds the target usage threshold, memory-based evictors start evicting entries and try to keep memory usage below the target usage threshold. However, no guarantee exists that the eviction speed is fast enough to avoid a potential out of memory error if the system runtime continues to quickly consume memory.

Writing a custom evictor

WebSphere eXtreme Scale allows you to write a custom eviction implementation.

You must create a custom evictor that implements the evictor interface and follows the common eXtreme Scale plug-in conventions. The interface follows:

```
public interface Evictor
{
    void initialize(BackingMap map, EvictionEventCallback callback);
    void activate();
    void apply(LogSequence sequence);
    void deactivate();
    void destroy();
}
```

- The initialize method is invoked during initialization of the BackingMap object. This method initializes an Evictor plug-in with a reference to the BackingMap and a reference to an object that implements the com.ibm.websphere.objectgrid.plugins.EvictionEventCallback interface.
- The activate method is called to activate the Evictor. After this method is called, the Evictor can use the EvictionEventCallback interface to evict map entries. If

the Evictor attempts to use the EvictionEventCallbackinterface to evict map entries before the activate method is called, an IllegalStateException exception results.

- The apply method is invoked when transactions that access one or more entries of the BackingMap are committed. The apply method is passed a reference to an object that implements the com.ibm.websphere.objectgrid.plugins.LogSequence interface. The LogSequence interface allows an Evictor plug-in to determine which BackingMap entries were created, modified, or removed by the transaction. An Evictor uses this information in deciding when and which entries to evict.
- The deactivate method is called to deactivate the Evictor. After this method is called, the Evictor must stop using the EvictionEventCallback interface to evict map entries. If the Evictor uses the EvictionEventcallback interface after this method is called, an IllegalStateException exception results.
- The destroy method is invoked when the BackingMap is being destroyed. This method allows an Evictor to terminate any threads that it might have created.

The EvictionEventCallback interface has the following methods:

```
public interface EvictionEventCallback
{
    void evictMapEntries(List evictorDataList) throws ObjectGridException;
    void evictEntries(List keysToEvictList) throws ObjectGridException;
    void setEvictorData(Object key, Object data);
    Object getEvictorData(Object key);
}
```

The EvictionEventCallback methods are used by an Evictor plug-in to call back to the eXtreme Scale framework as follows:

- The setEvictorData method is used by an evictor to request the framework that is used to store and associate some evictor object it creates with the entry indicated by the key argument. The data is evictor specific and is determined by the information the evictor needs to implement the algorithm it is using. For example, in a least frequently used algorithm, the evictor maintains a count in the evictor data object for tracking how many times the apply method is invoked with a LogElement that refers to an entry for a given key.
- The getEvictorData method is used by an evictor to retrieve the data it passed to the setEvictorData method during a prior apply method invocation. If evictor data for the specified key argument is not found, a special KEY_NOT_FOUND object that is defined on the EvictorCallback interface is returned.
- The evictMapEntries method is used by an evictor to request the eviction of one or more map entries. Each object in the evictorDataList parameter must implement the com.ibm.websphere.objectgrid.plugins.EvictorData interface. Also, the same EvictorData instance that is passed to the setEvictorData method must be in the evictor data list parameter of this method. The getKey method of the EvictorData interface is used to determine which map entry to evict. The map entry is evicted if the cache entry currently contains the exact same EvictorData instance that is in the evictor data list for this cache entry.
- The evictEntries method is used by an evictor to request eviction of one or more map entries. This method is used only if the object that is passed to the setEvictorData method does not implement the com.ibm.websphere.objectgrid.plugins.EvictorData interface.

After a transaction completes eXtreme Scale calls the apply method of the Evictor interface. All transaction locks that were acquired by the completed transaction are no longer held. Potentially, multiple threads can call the apply method at the same time, and each thread can complete its own transaction. Because transaction locks are already released by the completed transaction, the apply method must provide its own synchronization to ensure the apply method is thread safe.

The reason to implement the EvictorData interface and use the evictMapEntries method instead of the evictEntries method is to close a potential timing window. Consider the following sequence of events:

1. Transaction 1 completes and calls the apply method with a LogSequence that deletes the map entry for key 1.
2. Transaction 2 completes and calls the apply method with a LogSequence that inserts a new map entry for key 1. In other words, transaction 2 recreates the map entry that was deleted by transaction 1.

Because the evictor runs asynchronously from threads that run transactions, it is possible that when the evictor decides to evict key 1, it might be evicting either the map entry that existed prior to transaction 1 completion, or it might be evicting the map entry that was recreated by transaction 2. To eliminate timing windows and to eliminate uncertainty as to which version of the key 1 map entry the evictor intended to evict, implement the EvictorData interface by the object that is passed to the setEvictorData method. Use the same EvictorData instance for the life of a map entry. When that map entry is deleted and is then recreated by another transaction, the evictor should use a new instance of the EvictorData implementation. By using the EvictorData implementation and by using the evictMapEntries method, the evictor can ensure that the map entry is evicted if and only if the cache entry that is associated with the map entry contains the correct EvictorData instance.

The Evictor and EvictonEventCallback interfaces allow an application to plug in an evictor that implements a user-defined algorithm for eviction. The following snippet of code illustrates how you can implement the initialize method of Evictor interface:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
import java.util.LinkedList;
// Instance variables
private BackingMap bm;
private EvictionEventCallback evictorCallback;
private LinkedList queue;
private Thread evictorThread;
public void initialize(BackingMap map, EvictionEventCallback callback)
{
    bm = map;
    evictorCallback = callback;
    queue = new LinkedList();
    // spawn evictor thread
    evictorThread = new Thread( this );
    String threadName = "MyEvictorForMap-" + bm.getName();
    evictorThread.setName( threadName );
    evictorThread.start();
}
```

The preceding code saves the references to the map and callback objects in instance variables so that they are available to the apply and destroy methods. In this example, a linked list is created that is used as a first in, first out queue for implementing a least recently used (LRU) algorithm. A thread is spawned off and a reference to the thread is kept as an instance variable. By keeping this reference, the destroy method can interrupt and terminate the spawned thread.

Ignoring synchronization requirements to make code thread safe, the following snippet of code illustrates how the apply method of the Evictor interface can be implemented:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.EvictorData;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;

public void apply(LogSequence sequence)
{
    Iterator iter = sequence.getAllChanges();
    while ( iter.hasNext() )
    {
        LogElement elem = (LogElement)iter.next();
        Object key = elem.getKey();
        LogElement.Type type = elem.getType();
        if ( type == LogElement.INSERT )
        {
            // do insert processing here by adding to front of LRU queue.
            EvictorData data = new EvictorData(key);
            evictorCallback.setEvictorData(key, data);
            queue.addFirst( data );
        }
        else if ( type == LogElement.UPDATE || type == LogElement.FETCH || type == LogElement.TOUCH )
        {
            // do update processing here by moving EvictorData object to
            // front of queue.
            EvictorData data = evictorCallback.getEvictorData(key);
            queue.remove(data);
            queue.addFirst(data);
        }
        else if ( type == LogElement.DELETE || type == LogElement.EVICT )
        {
            // do remove processing here by removing EvictorData object
            // from queue.
            EvictorData data = evictorCallback.getEvictorData(key);
            if ( data == EvictionEventCallback.KEY_NOT_FOUND )
            {
                // Assumption here is your asynchronous evictor thread
                // evicted the map entry before this thread had a chance
                // to process the LogElement request. So you probably
                // need to do nothing when this occurs.
            }
            else
            {
                // Key was found. So process the evictor data.
                if ( data != null )
                {
                    // Ignore null returned by remove method since spawned
                    // evictor thread may have already removed it from queue.
                    // But we need this code in case it was not the evictor
                    // thread that caused this LogElement to occur.
                    queue.remove( data );
                }
                else
                {
                    // Depending on how you write you Evictor, this possibility
                    // may not exist or it may indicate a defect in your evictor
                    // due to improper thread synchronization logic.
                }
            }
        }
    }
}
```

Insert processing in the apply method typically handles the creation of an evictor data object that is passed to the setEvictorData method of the EvictionEventCallback interface. Because this evictor illustrates a LRU implementation, the EvictorData is also added to the front of the queue that was created by the initialize method. Update processing in the apply method typically updates the evictor data object that was created by some prior invocation of the apply method (for example, by the insert processing of the apply method). Because this evictor is an LRU implementation, it needs to move the EvictorData object from its current queue position to the front of the queue. The spawned evictor thread removes the last EvictorData object in the queue because the last queue element represents the least recently used entry. The assumption is that the EvictorData object has a getKey method on it so that the evictor thread knows the keys of the entries that need to be evicted. Keep in mind that this example is ignoring synchronization requirements to make code thread safe. A real custom evictor is more complicated because it deals with synchronization and performance bottlenecks that occur as a result of the synchronization points.

The following snippets of code illustrate the destroy method and the run method of the runnable thread that the initialize method spawned:

```
// Destroy method simply interrupts the thread spawned by the initialize method.
public void destroy()
{
    evictorThread.interrupt();
}

// Here is the run method of the thread that was spawned by the initialize method.
public void run()
{
    // Loop until destroy method interrupts this thread.
    boolean continueToRun = true;
    while ( continueToRun )
    {
        try
        {
            // Sleep for a while before sweeping over queue.
            // The sleepTime is a good candidate for a evictor
            // property to be set.
            Thread.sleep( sleepTime );
            int queueSize = queue.size();
            // Evict entries if queue size has grown beyond the
            // maximum size. Obviously, maximum size would
            // be another evictor property.
            int numToEvict = queueSize - maxSize;
            if ( numToEvict > 0 )
            {
                // Remove from tail of queue since the tail is the
                // least recently used entry.
                List evictList = new ArrayList( numToEvict );
                while( queueSize > ivMaxSize )
                {
                    EvictorData data = null;
                    try
                    {
                        EvictorData data = (EvictorData) queue.removeLast();
                        evictList.add( data );
                        queueSize = queue.size();
                    }
                    catch ( NoSuchElementException nse )
                    {
                        // The queue is empty.
                        queueSize = 0;
                    }
                }
                // Request eviction if key list is not empty.
                if ( ! evictList.isEmpty() )
                {
                    evictorCallback.evictMapEntries( evictList );
                }
            }
        }
        catch ( InterruptedException e )
        {
            continueToRun = false;
        }
    } // end while loop
} // end run method.
```

Optional RollBackEvictor interface

The com.ibm.websphere.objectgrid.plugins.RollbackEvictor interface can be optionally implemented by an Evictor plug-in. By implementing this interface, an evictor can be invoked not only when transactions are committed, but also when transactions are rolled back.

```
public interface RollbackEvictor
{
    void rollingBack( LogSequence ls );
}
```

The apply method is called only if a transaction is committed. If a transaction is rolled back and the RollbackEvictor interface is implemented by the evictor, the rollingBack method is invoked. If the RollbackEvictor interface is not implemented and the transaction rolls back, the apply method and the rollingBack method are not called.

Writing an index plug-in

With a MapIndexPlugin plug-in, or index, you can write custom indexing strategies that are beyond the built-in indexes that eXtreme Scale provides.

For general information about indexing, see “Indexing” on page 118.

For information about using indexing, see “Using indexing for non-key data access” on page 120.

MapIndexPlugin implementations must use the MapIndexPlugin interface and follow the common eXtreme Scale plug-in conventions.

The following sections include some of the important methods of the index interface.

setProperties method

Use the setProperties method to initialize the index plug-in programmatically. The Properties object parameter that is passed into the method should contain required configuration information to initialize the index plug-in properly. The setProperties method implementation, along with the getProperties method implementation, are required in a distributed environment because the index plug-in configuration moves between client and server processes. An implementation example of this method follows.

```
setProperties(Properties properties)

// setProperties method sample code
    public void setProperties(Properties properties) {
        ivIndexProperties = properties;

        String ivRangeIndexString = properties.getProperty("rangeIndex");
        if (ivRangeIndexString != null && ivRangeIndexString.equals("true")) {
            setRangeIndex(true);
        }
        setName(properties.getProperty("indexName"));
        setAttributeName(properties.getProperty("attributeName"));

        String ivFieldAccessAttributeString = properties.getProperty("fieldAccessAttribute");
        if (ivFieldAccessAttributeString != null && ivFieldAccessAttributeString.equals("true")) {
            setFieldAccessAttribute(true);
        }

        String ivPOJOKeyIndexString = properties.getProperty("POJOKeyIndex");
        if (ivPOJOKeyIndexString != null && ivPOJOKeyIndexString.equals("true")) {
            setPOJOKeyIndex(true);
        }
    }
```

getProperties method

The getProperties method extracts the index plug-in configuration from a MapIndexPlugin instance. You can use the extracted properties to initialize another MapIndexPlugin instance to have the same internal states. The getProperties method and setProperties method implementations are required in a distributed environment. An implementation example of the getProperties method follows.

```
getProperties()

// getProperties method sample code
    public Properties getProperties() {
        Properties p = new Properties();
        p.put("indexName", indexName);
        p.put("attributeName", attributeName);
        p.put("rangeIndex", ivRangeIndex ? "true" : "false");
        p.put("fieldAccessAttribute", ivFieldAccessAttribute ? "true" : "false");
        p.put("POJOKeyIndex", ivPOJOKeyIndex ? "true" : "false");
        return p;
    }
```

setEntityMetadata method

The setEntityMetadata method is called by the WebSphere eXtreme Scale run time during initialization to set the EntityMetadata of the associated BackingMap on the MapIndexPlugin instance. The EntityMetadata is required for supporting indexing of tuple objects. A tuple is a data set that represents an entity object or its key. If the BackingMap is for an entity, then you must implement this method.

The following code sample implements the setEntityMetadata method.

```
setEntityMetadata(EntityMetadata entityMetadata)
```

```
// setEntityMetadata method sample code
    public void setEntityMetadata(EntityMetadata entityMetadata) {
        ivEntityMetadata = entityMetadata;
        if (ivEntityMetadata != null) {
            // this is a tuple map
            TupleMetadata valueMetadata = ivEntityMetadata.getValueMetadata();
            int numAttributes = valueMetadata.getNumAttributes();
            for (int i = 0; i < numAttributes; i++) {
                String tupleAttributeName = valueMetadata.getAttribute(i).getName();
                if (attributeName.equals(tupleAttributeName)) {
                    ivTupleValueIndex = i;
                    break;
                }
            }

            if (ivTupleValueIndex == -1) {
                // did not find the attribute in value tuple, try to find it on key tuple.
                // if found on key tuple, implies key indexing on one of tuple key attributes.
                TupleMetadata keyMetadata = ivEntityMetadata.getKeyMetadata();
                numAttributes = keyMetadata.getNumAttributes();
                for (int i = 0; i < numAttributes; i++) {
                    String tupleAttributeName = keyMetadata.getAttribute(i).getName();
                    if (attributeName.equals(tupleAttributeName)) {
                        ivTupleValueIndex = i;
                        ivKeyTupleAttributeIndex = true;
                        break;
                    }
                }
            }

            if (ivTupleValueIndex == -1) {
                // if entityMetadata is not null and we could not find the
        // attributeName in entityMetadata, this is an
                // error
                throw new ObjectGridRuntimeException("Invalid attributeName.  Entity: " +
         ivEntityMetadata.getName());
            }
        }
    }
```

Attribute name methods

The setAttributeName method sets the name of the attribute to be indexed. The cached object class must provide the get method for the indexed attribute. For example, if the object has an employeeName or EmployeeName attribute, the index calls the getEmployeeName method on the object to extract the attribute value. The attribute name must be the same as the name in the get method, and the attribute must implement the Comparable interface. If the attribute is boolean type, you can also use the isAttributeName method pattern.

The getAttributeName method returns the name of the indexed attribute.

getAttribute method

The getAttribute method returns the indexed attribute value from the specified object. For example, if an Employee object has an attribute called employeeName that is indexed, you can use the getAttribute method to extract the employeeName attribute value from a specified Employee object. This method is required in a distributed WebSphere eXtreme Scale environment.

```
getAttribute(Object value)

// getAttribute method sample code
    public Object getAttribute(Object value) throws ObjectGridRuntimeException {
        if (ivPOJOKeyIndex) {
            // In the POJO key indexing case, no need to get attribute from value object.
            // The key itself is the attribute value used to build the index.
            return null;
        }

        try {
            Object attribute = null;
            if (value != null) {
                // handle Tuple value if ivTupleValueIndex != -1
                if (ivTupleValueIndex == -1) {
                    // regular value
                    if (ivFieldAccessAttribute) {
                        attribute = this.getAttributeField(value).get(value);
                    } else {
                        attribute = getAttributeMethod(value).invoke(value, emptyArray);
                    }
                } else {
                    // Tuple value
                    attribute = extractValueFromTuple(value);
                }
            }
            return attribute;
        } catch (InvocationTargetException e) {
             throw new ObjectGridRuntimeException(
                    "Caught unexpected Throwable during index update processing,
           index name = " + indexName + ": " + t,
                    t);
        } catch (Throwable t) {
             throw new ObjectGridRuntimeException(
                    "Caught unexpected Throwable during index update processing,
          index name = " + indexName + ": " + t,
                    t);
        }
    }
```

TransactionCallback plug-in

Use the TransactionCallback plug-in to customize versioning and comparison operations of cache objects when you are using the optimistic locking strategy.

You can provide a pluggable optimistic callback object that implements the com.ibm.websphere.objectgrid.plugins.OptimisticCallback interface. For entity maps, a high performance OptimisticCallback plug-in is automatically configured.

Purpose

Use the OptimisticCallback interface to provide optimistic comparison operations for the values of a map. An OptimisticCallback implementation is required when you use the optimistic locking strategy. WebSphere eXtreme Scale provides a default OptimisticCallback implementation. However, usually the application must plug in its own implementation of the OptimisticCallback interface. See the information about locking strategies in the *Product Overview* for more information.

Default implementation

The eXtreme Scale framework provides a default implementation of the OptimisticCallback interface that is used if the application does not plug in an application-provided OptimisticCallback object, as demonstrated in the previous section. The default implementation always returns the special value of NULL_OPTIMISTIC_VERSION as the version object for the value and never updates the version object. This action makes optimistic comparison a no operation function. In most cases, you do not want the no operation function to occur when you are using the optimistic locking strategy. Your applications must implement the OptimisticCallback interface and plug in their own OptimisticCallback implementations so that the default implementation is not used. However, at least one scenario exists where the default provided OptimisticCallback implementation is useful. Consider the following situation:

- A loader is plugged for the backing map.
- The loader knows how to perform the optimistic comparison without assistance from an OptimisticCallback plug-in.

How can the loader know how to deal with optimistic versioning without assistance from an OptimisticCallback object? The loader has knowledge of the value class object and knows which field of the value object is used as an optimistic versioning value. For example, suppose the following interface is used for the value object for the employees map:

```
public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}
```

In this case, the loader knows that it can use the getSequenceNumber method to get the current version information for an Employee value object. The loader increments the returned value to generate a new version number before updating the persistent storage with the new Employee value. For a Java database connectivity (JDBC) loader, the current sequence number in the where clause of an overqualified SQL update statement is used, and it uses the new generated sequence number to set the sequence number column to the new sequence number value.

Another possibility is that the loader makes use of some backend-provided function that automatically updates a hidden column that can be used for optimistic versioning. In some cases, a stored procedure or trigger can possibly be used to help maintain a column that holds versioning information. If the loader is using one of these techniques for maintaining optimistic versioning information, then the application does not need to provide an OptimisticCallback implementation. You can use the default OptimisticCallback implementation because the loader is able to handle optimistic versioning without any assistance from an OptimisticCallback object.

Default implementation for entities

Entities are stored in the ObjectGrid using tuple objects. The default OptimisticCallback implementation behaves similarly to the behavior for non-entity maps. However, the version field in the entity is identified using the @Version annotation or the version attribute in the entity descriptor XML file.

The version attribute can be of the following types: int, Integer, short, Short, long, Long or java.sql.Timestamp. An entity should have only one version attribute defined. The version attribute should be set only during construction. After the entity is persisted, the value of the version attribute should not be modified.

If a version attribute is not configured and the optimistic locking strategy is used, then the entire tuple is implicitly versioned using the entire state of the tuple.

In the following example, the Employee entity has a long version attribute named SequenceNumber:

```
@Entity
public class Employee
{
private long sequence;
    // Sequential sequence number used for optimistic versioning.
    @Version
    public long getSequenceNumber() {
        return sequence;
    }
    public void setSequenceNumber(long newSequenceNumber) {
        this.sequence = newSequenceNumber;
    }
    // Other get/set methods for other fields of Employee object.
}
```

Writing an OptimisticCallback implementation

An OptimisticCallback implementation must implement the OptimisticCallback interface and follow the common ObjectGrid plug-in conventions

The following list provides a description or consideration for each of the methods in the OptimisticCallback interface:

NULL_OPTIMISTIC_VERSION

This special value is returned by getVersionedObjectForValue method if the default OptimisticCallback implementation is used instead of an application-provided OptimisticCallback implementation.

getVersionedObjectForValue method

The getVersionedObjectForValue method might return a copy of the value or it might return an attribute of the value that can be used for versioning purposes. This method is called whenever an object is associated with a transaction. When no Loader is set into a backing map, the backing map uses this value at commit time to perform an optimistic version comparison. The optimistic version comparison is used by the backing map to ensure that the version has not changed since this transaction first accessed the map entry that was modified by this transaction. If another transaction had already modified the version for this map entry, the version comparison fails and the backing map displays an OptimisticCollisionException exception to force rollback of the transaction. If a Loader is plugged in, the backing map does not use the optimistic versioning information. Instead, the Loader is responsible for performing the optimistic versioning comparison and updating the versioning information when necessary. The Loader typically gets the initial versioning object from the LogElement passed to the batchUpdate method on the Loader, which is called when a flush operation occurs or a transaction is committed.

The following code shows the implementation used by the EmployeeOptimisticCallbackImpl object:

```
public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}
```

As demonstrated in the previous example, the sequenceNumber attribute is returned in a java.lang.Long object as expected by the Loader, which implies that the same person that wrote the Loader either wrote the EmployeeOptimisticCallbackImpl implementation or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl implementation. For example, these people agreed on the value that is returned by the getVersionedObjectForValue method. As previously described, the default OptimisticCallback implementation returns the special value NULL_OPTIMISTIC_VERSION as the version object.

updateVersionedObjectForValue method

The updateVersionedObjectForValue method is called when a transaction has updated a value and a new versioned object is needed. If the getVersionedObjectForValue method returns an attribute of the value, this method typically updates the attribute value with a new version object. If the getVersionedObjectForValue method returns a copy of the value, this method typically would not update. The default OptimisticCallback does not update because the default implementation of the getVersionedObjectForValue method always returns the special value NULL_OPTIMISTIC_VERSION as the version object. The following example shows the implementation used by the EmployeeOptimisticCallbackImpl object that is used in the OptimisticCallback section:

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

As demonstrated in the previous example, the sequenceNumber attribute is increments by one so that the next time the getVersionedObjectForValue method is called, the java.lang.Long value that is returned has a long value that is the original sequence number value. Plus one, for example, is the next version value for this employee instance. Again, this example implies that the same person that wrote the Loader either wrote EmployeeOptimisticCallbackImpl implementation or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl implementation.

serializeVersionedValue method

This method writes the versioned value to the specified stream. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the proper serialization. The default implementation calls the writeObject method.

inflateVersionedValue method

This method takes the serialized version of the versioned value and returns the actual versioned value object. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the proper deserialization. The default implementation calls the readObject method.

Using an application-provided OptimisticCallback implementation

You have two approaches to add an application-provided OptimisticCallback into the BackingMap configuration: programmatic configuration and XML configuration.

Programmatically plug in an OptimisticCallback implementation

The following example demonstrates how an application can programmatically plug in an OptimisticCallback object for the employee backing map in the grid1 ObjectGrid instance:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

XML configuration approach to plug in an OptimisticCallback implementation

The EmployeeOptimisticCallbackImpl object in the preceding example must implement the OptimisticCallback interface. The application can also use an XML file to plug in its OptimisticCallback object as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid1">
        <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>

<backingMapPluginCollections>
    <backingMapPluginCollection id="employees">
        <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

Introduction to plug-in slots

A plug-in slot is a transactional storage space that is reserved for plug-ins that share transactional context. These slots provide a way for eXtreme Scale plug-ins to communicate with each other, share transactional context, and ensure that transactional resources are used correctly and consistently within a transaction.

A plug-in can store transactional context, such as database connection, Java Message Service (JMS) connection, and so on, in a plug-in slot. The stored transactional context can be retrieved by any plug-in that knows the plug-in slot number, which serves as the key to retrieve transactional context.

Using plug-in slots

Plug-in slots are part of the TxID Interface. See the API documentation for more information about the interface.The slots are entries on an ArrayList array. Plug-ins can reserve an entry in the ArrayList array by calling the ObjectGrid.reserveSlot method and indicating that it wants a slot on all TxID objects. After the slots are reserved, plug-ins can put transactional context into slots of each TxID object and retrieve the context later. The put and get operations are coordinated by slot numbers that are returned by the ObjectGrid.reserveSlot method.

A plug-in typically has a life cycle. Using plug-in slots has to fit into the life cycle of plug-in. Typically, the plug-in must reserve plug-in slots during the initialization stage and obtain slot numbers for each slot. During normal run time, the plug-in puts transactional context into the reserved slot in the TxID object at the appropriate point. This appropriate point is typically at the beginning of the transaction. The plug-in or other plug-ins can then get the stored transactional context by the slot number from the TxID within the transaction.

The plug-in typically performs a clean up by removing transactional context and the slots. The following snippet of code illustrates how to use plug-in slots in a TransactionCallback plug-in:

```
public class DatabaseTransactionCallback implements TransactionCallback {
    int connectionSlot;
    int autoCommitConnectionSlot;
    int psCacheSlot;
    Properties ivProperties = new Properties();

    public void initialize(ObjectGrid objectGrid) throws TransactionCallbackException {
        // In initialization stage, reserve desired plug-in slots by calling the
        //reserveSlot method of ObjectGrid and
        // passing in the designated slot name, TxID.SLOT_NAME.
        // Note: you have to pass in this TxID.SLOT_NAME that is designated
        // for application.
        try {
            // cache the returned slot numbers
            connectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            psCacheSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            autoCommitConnectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
        } catch (Exception e) {
        }
    }

    public void begin(TxID tx) throws TransactionCallbackException {
        // put transactional contexts into the reserved slots at the
        // beginning of the transaction.
        try {
            Connection conn = null;
            conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
            tx.putSlot(connectionSlot, conn);
            conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
            conn.setAutoCommit(true);
            tx.putSlot(autoCommitConnectionSlot, conn);
            tx.putSlot(psCacheSlot, new HashMap());
        } catch (SQLException e) {
            SQLException ex = getLastSQLException(e);
            throw new TransactionCallbackException("unable to get connection", ex);
        }
    }

    public void commit(TxID id) throws TransactionCallbackException {
        // get the stored transactional contexts and use them
        // then, clean up all transactional resources.
        try {
            Connection conn = (Connection) id.getSlot(connectionSlot);
            conn.commit();
            cleanUpSlots(id);
        } catch (SQLException e) {
            SQLException ex = getLastSQLException(e);
            throw new TransactionCallbackException("commit failure", ex);
        }
    }

    void cleanUpSlots(TxID tx) throws TransactionCallbackException {
        closePreparedStatements((Map) tx.getSlot(psCacheSlot));
        closeConnection((Connection) tx.getSlot(connectionSlot));
        closeConnection((Connection) tx.getSlot(autoCommitConnectionSlot));
    }

    /**
     * @param map
     */
    private void closePreparedStatements(Map psCache) {
        try {
            Collection statements = psCache.values();
            Iterator iter = statements.iterator();
            while (iter.hasNext()) {
                PreparedStatement stmt = (PreparedStatement) iter.next();
                stmt.close();
            }
        } catch (Throwable e) {
        }

    }

    /**
     * Close connection and swallow any Throwable that occurs.
     *
     * @param connection
     */
    private void closeConnection(Connection connection) {
        try {
            connection.close();
        } catch (Throwable e1) {
        }
    }

    public void rollback(TxID id) throws TransactionCallbackException
        // get the stored transactional contexts and use them
        // then, clean up all transactional resources.
        try {
            Connection conn = (Connection) id.getSlot(connectionSlot);
            conn.rollback();
            cleanUpSlots(id);
        } catch (SQLException e) {
        }
    }

    public boolean isExternalTransactionActive(Session session) {
        return false;
    }

    // Getter methods for the slot numbers, other plug-in can obtain the slot numbers
    // from these getter methods.

    public int getConnectionSlot() {
        return connectionSlot;
    }
    public int getAutoCommitConnectionSlot() {
        return autoCommitConnectionSlot;
    }
    public int getPreparedStatementSlot() {
        return psCacheSlot;
    }
```

The following snippet of code illustrates how a Loader can get the stored transactional context that is put by previous TransactionCallback plug-in example:

```
public class DatabaseLoader implements Loader
{
    DatabaseTransactionCallback tcb;
    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
    {
        // The preload method is the initialization method of the Loader.
        // Obtain interested plug-in from Session or ObjectGrid instance.
        tcb =
   (DatabaseTransactionCallback)session.getObjectGrid().getTransactionCallback();
    }
    public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement get here
        return null;
    }
    public void batchUpdate(TxID txid, LogSequence sequence) throws LoaderException,
   OptimisticCollisionException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement batch update here ...
    }
}
```

External transaction managers

Typically, eXtreme Scale transactions begin with the Session.begin method and end with the Session.commit method. However, when an ObjectGrid is embedded, an external transaction coordinator can start and end transactions. In this case, you do not need to call the begin or commit methods.

External transaction coordination

The TransactionCallback plug-in is extended with the isExternalTransactionActive(Session session) method that associates the eXtreme Scale session with an external transaction. The method header follows:

```
public synchronized boolean isExternalTransactionActive(Session session)
```

For example, eXtreme Scale can be set up to integrate with WebSphere Application Server and WebSphere Extended Deployment.

Also, eXtreme Scale provides a built in plug-in called the WebSphere "TransactionCallback plug-in" on page 148, which describes how to build the plug-in for WebSphere Application Server environments, but you can adapt the plug-in for other frameworks.

The key to this seamless integration is the exploitation of the ExtendedJTATransaction API in WebSphere Application Server Version 5.x and Version 6.x. However, if you are using WebSphere Application Server Version 6.0.2, you must apply APAR PK07848 to support this method. Use the following sample code to associate an ObjectGrid session with a WebSphere Application Server transaction ID:

```
/**
* This method is required to associate an objectGrid session with a WebSphere
* Application Server transaction ID.
*/
Map/**/ localIdToSession;
public synchronized boolean isExternalTransactionActive(Session session)
{
    // remember that this localid means this session is saved for later.
    localIdToSession.put(new Integer(jta.getLocalId()), session);
    return true;
}
```

Retrieve an external transaction

Sometimes you might need to retrieve an external transaction service object for the TransactionCallback plug-in to use. In the WebSphere Application Server server, look up the ExtendedJTATransaction object from its namespace as shown in the following example:

```
public J2EETransactionCallback() {
    super();
    localIdToSession = new HashMap();
    String lookupName="java:comp/websphere/ExtendedJTATransaction";
    try
    {
        InitialContext ic = new InitialContext();
        jta = (ExtendedJTATransaction)ic.lookup(lookupName);
        jta.registerSynchronizationCallback(this);
    }
    catch(NotSupportedException e)
    {
        throw new RuntimeException("Cannot register jta callback", e);
    }
    catch(NamingException e){
        throw new RuntimeException("Cannot get transaction object");
    }
}
```

For other products, you can use a similar approach to retrieve the transaction service object.

Control commit by external callback

The TransactionCallback plug-in must receive an external signal to commit or roll back the eXtreme Scale session. To receive this external signal, use the callback from the external transaction service. Implement the external callback interface and register it with the external transaction service. For example, with WebSphere Application Server, implement the SynchronizationCallback interface, as shown in the following example:

```
public class J2EETransactionCallback implements
 com.ibm.websphere.objectgrid.plugins.TransactionCallback, SynchronizationCallback {
   public J2EETransactionCallback() {
      super();
      String lookupName="java:comp/websphere/ExtendedJTATransaction";
      localIdToSession = new HashMap();
      try {
         InitialContext ic = new InitialContext();
         jta = (ExtendedJTATransaction)ic.lookup(lookupName);
         jta.registerSynchronizationCallback(this);
      } catch(NotSupportedException e) {
         throw new RuntimeException("Cannot register jta callback", e);
      }
      catch(NamingException e) {
         throw new RuntimeException("Cannot get transaction object");
      }
   }

   public synchronized void afterCompletion(int localId, byte[] arg1,boolean didCommit) {
      Integer lid = new Integer(localId);
      // find the Session for the localId
      Session session = (Session)localIdToSession.get(lid);
      if(session != null) {
          try {
            // if WebSphere Application Server is committed when
            // hardening the transaction to backingMap.
            // We already did a flush in beforeCompletion
            if(didCommit) {
               session.commit();
            } else {
               // otherwise rollback
               session.rollback();
            }
         } catch(NoActiveTransactionException e) {
            // impossible in theory
         } catch(TransactionException e) {
            // given that we already did a flush, this should not fail
         } finally {
            // always clear the session from the mapping map.
            localIdToSession.remove(lid);
         }
      }
   }

   public synchronized void beforeCompletion(int localId, byte[] arg1) {
      Session session = (Session)localIdToSession.get(new Integer(localId));
      if(session != null) {
         try {
            session.flush();
         } catch(TransactionException e) {
            // WebSphere Application Server  does not formally define
            // a way to signal the
            // transaction has failed so do this
            throw new RuntimeException("Cache flush failed", e);
         }
      }
   }
}
```

Use eXtreme Scale APIs with the TransactionCallback plug-in

The TransactionCallback plug-in disables autocommit in eXtreme Scale. The normal usage pattern for an eXtreme Scale follows:

```
Session ogSession = ...;
ObjectMap myMap = ogSession.getMap("MyMap");
ogSession.begin();
MyObject v = myMap.get("key");
v.setAttribute("newValue");
myMap.update("key", v);
ogSession.commit();
```

When this TransactionCallback plug-in is in use, eXtreme Scale assumes that the application uses the eXtreme Scale when a container-managed transaction is present. The previous code snippet changes the following code in this environment:

```
public void myMethod() {
   UserTransaction tx = ...;
   tx.begin();
   Session ogSession = ...;
   ObjectMap myMap = ogSession.getMap("MyMap");
   yObject v = myMap.get("key");
   v.setAttribute("newValue");
   myMap.update("key", v);
   tx.commit();
}
```

The myMethod method is similar to a Web application scenario. The application uses the normal UserTransaction interface to begin, commit, and roll back transactions. The eXtreme Scale automatically begins and commits around the container transaction. If the method is an Enterprise JavaBeans (EJB) method that uses the TX_REQUIRES attribute, then remove the UserTransaction reference and the calls to begin and commit transactions and the method works the same way. In this case, the container is responsible for starting and ending the transaction.

Using a Loader

With an eXtreme Scale Loader plug-in, an ObjectGrid map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or some other system. Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data, allowing hub-based caches to be built using ObjectGrid. A loader has the logic for reading and writing data to and from a persistent store.

Loaders are backing map plug-ins that are invoked when changes are made to the backing map or when the backing map is unable to satisfy a data request (a cache miss).

See the information about caching concepts in the *Product Overview* for more information.

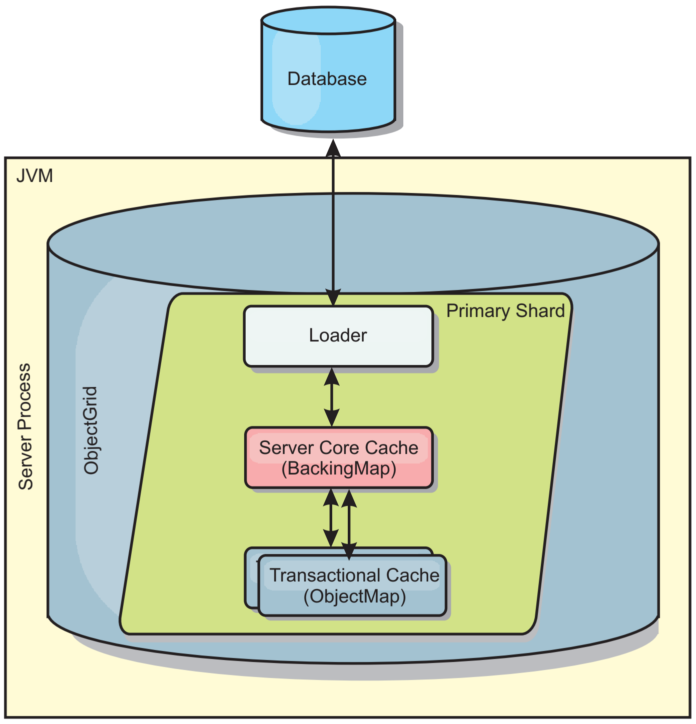
Database
JVM
Server Process
ObjectGrid
Primary Shard
Loader
Server Core Cache (BackingMap)
Transactional Cache (ObjectMap)

Figure 3. Loader

WebSphere eXtreme Scale includes two built-in loaders to integrate with relational database back ends. The Java Persistence API (JPA) loaders use the Object-Relational Mapping (ORM) capabilities of both the OpenJPA and Hibernate implementations of the JPA specification.

Using a loader

To add a loader into the BackingMap configuration, you can use programmatic configuration or XML configuration. A loader has the following relationship with a backing map:

- A backing map can have only one loader.
- A client backing map (near cache) cannot have a loader.
- A loader definition can be applied to multiple backing maps, but each backing map has its own loader instance.

Programmatically plug in a Loader

The following snippet of code demonstrates how to plug an application-provided Loader into the backing map for map1 using the ObjectGrid API:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
MyLoader loader = new MyLoader();
loader.setDataBaseName("testdb");
loader.setIsolationLevel("read committed");
bm.setLoader( loader );
```

This snippet assumes that the MyLoader class is the application-provided class that implements the com.ibm.websphere.objectgrid.plugins.Loader interface. Because the association of a Loader with a backing map cannot be changed after ObjectGrid is initialized, the code must be run before invoking the initialize method of the ObjectGrid interface that is being called. An IllegalStateException exception occurs on a setLoader method call if it is called after initialization has occurred.

The application-provided Loader can have set properties. In the example, the MyLoader loader is used to read and write data from a table in a relational database. The loader must specify the name of the database and the SQL isolation level. The MyLoader loader has the setDataBaseName and setIsolationLevel methods that allow the application to set these two Loader properties.

XML configuration approach to plug in a Loader

An application-provided Loader can also be plugged in by using an XML file. The following example demonstrates how to plug the MyLoader loader into the map1 backing map with the same database name and isolation level Loader properties:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid">
        <backingMap name="map1" pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>
<backingMapPluginCollections>
    <backingMapPluginCollection id="map1">
        <bean id="Loader" className="com.myapplication.MyLoader">
            <property name="dataBaseName"
                      type="java.lang.String"
                      value="testdb"
                      description="database name" />
            <property name="isolationLevel"
                      type="java.lang.String"
                      value="read committed"
                      description="iso level" />
        </bean>
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

Related tasks

"Troubleshooting loaders" on page 274
Use this information to troubleshoot issues with your database loaders.

Related reference

"JPA loader programming considerations" on page 164
A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

Writing a loader

You can write your own loader plug-in implementation in your applications, which must follow the common WebSphere eXtreme Scale plug-in conventions.

Including a loader plug-in

The Loader interface has the following definition:

```
public interface Loader
{
    static final SpecialValue KEY_NOT_FOUND;
    List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException;
    void batchUpdate(TxID txid, LogSequence sequence) throws
    LoaderException, OptimisticCollisionException;
    void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
}
```

See the information about Loaders in the *Administration Guide* for more information.

get method

The backing map calls the Loader get method to get the values that are associated with a key list that is passed as the keyList argument. The get method is required to return a java.lang.util.List list of values, one value for each key that is in the key list. The first value that is returned in the value list corresponds to the first key in the key list, the second value returned in the value list corresponds to the second key in the key list, and so on. If the loader does not find the value for a key in the key list, the Loader is required to return the special KEY_NOT_FOUND value object that is defined in the Loader interface. Because a backing map can be configured to allow `null` as a valid value, it is very important for the Loader to return the special KEY_NOT_FOUND object when the Loader cannot find the key. This special value allows the backing map to distinguish between a null value and a value that does not exist because the key was not found. If a backing map does not support null values, a Loader that returns a null value instead of the KEY_NOT_FOUND object for a key that does not exist results in an exception.

The forUpdate argument tells the Loader if the application called a get method on the map or a getForUpdate method on the map. See the ObjectMap interface in the API documentation for more information. The Loader is responsible for implementing a concurrency control policy that controls concurrent access to the persistent store. For example, many relational database management systems support the for update syntax on the SQL select statement that is used to read data from a relational table. The Loader can choose to use the for update syntax on the SQL select statement based on whether `boolean true` is passed as the argument value for the forUpdate parameter of this method. Typically, the Loader uses the for update syntax only when the pessimistic concurrency control policy is used. For an optimistic concurrency control, the Loader never uses `for update` syntax on the SQL select statement. The Loader is responsible to decide to use the forUpdate argument based on the concurrency control policy that is being used by the Loader.

For an explanation of the txid parameter, see “TransactionCallback plug-in” on page 148.

batchUpdate method

The batchUpdate method is important on the Loader interface. This method is called whenever the eXtreme Scale needs to apply all the current changes to the Loader. The Loader is given a list of changes for the selected Map. The changes are iterated and applied to the backend. The method receives the current TxID value and the changes to apply. The following sample iterates over the set of changes and batches three Java database connectivity (JDBC) statements, one with insert, another with update, and one with delete.

```
import java.util.Collection;
import java.util.Map;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;

    public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException {
        // Get a SQL connection to use.
        Connection conn = getConnection(tx);
        try {
            // Process the list of changes and build a set of prepared
            // statements for executing a batch update, insert, or delete
            // SQL operation.
            Iterator iter = sequence.getPendingChanges();
            while (iter.hasNext()) {
                LogElement logElement = (LogElement) iter.next();
                Object key = logElement.getKey();
                Object value = logElement.getCurrentValue();
                switch (logElement.getType().getCode()) {
                case LogElement.CODE_INSERT:
                    buildBatchSQLInsert(tx, key, value, conn);
                    break;
                case LogElement.CODE_UPDATE:
                    buildBatchSQLUpdate(tx, key, value, conn);
                    break;
                case LogElement.CODE_DELETE:
                    buildBatchSQLDelete(tx, key, conn);
                    break;
                }
            }
            // Execute the batch statements that were built by above loop.
            Collection statements = getPreparedStatementCollection(tx, conn);
            iter = statements.iterator();
            while (iter.hasNext()) {
                PreparedStatement pstmt = (PreparedStatement) iter.next();
                pstmt.executeBatch();
            }
        } catch (SQLException e) {
            LoaderException ex = new LoaderException(e);
            throw ex;
        }
    }
```

The preceding sample illustrates the high level logic of processing the LogSequence argument, but the details of how a SQL insert, update, or delete statement is built are not illustrated. Some of the key points that are illustrated include:

- The getPendingChanges method is called on the LogSequence argument to obtain an iterator over the list of LogElements that the Loader needs to process.
- The LogElement.getType().getCode() method is used to determine if the LogElement is for a SQL insert, update, or delete operation.
- An SQLException exception is caught and is chained to a LoaderException exception that prints to report that an exception occurred during the batch update.
- JDBC batch update support is used to minimize the number of queries to the backend that must be made.

preloadMap method

During the eXtreme Scale initialization, each backing map that is defined is initialized. If a Loader is plugged into a backing map, the backing map invokes the preloadMap method on the Loader interface to allow the loader to prefetch data from its back end and load the data into the map. The following sample assumes the first 100 rows of an Employee table is read from the database and is loaded into the map. The EmployeeRecord class is an application-provided class that holds the employee data that is read from the employee table.

Note: This sample fetches all the data from database and then insert them into the base map of one partition. In a real-world distributed eXtreme Scale deployment scenario, data should be distributed into all the partitions. Refer to “Client-based JPA preload utility programming” on page 182 for more information.

```
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException

    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException {
        boolean tranActive = false;
        ResultSet results = null;
        Statement stmt = null;
        Connection conn = null;
        try {
            session.beginNoWriteThrough();
            tranActive = true;
            ObjectMap map = session.getMap(backingMap.getName());
            TxID tx = session.getTxID();
            // Get a auto-commit connection to use that is set to
            // a read committed isolation level.
            conn = getAutoCommitConnection(tx);
            // Preload the Employee Map with EmployeeRecord
            // objects. Read all Employees from table, but
            // limit preload to first 100 rows.
            stmt = conn.createStatement();
            results = stmt.executeQuery(SELECT_ALL);
            int rows = 0;
            while (results.next() && rows < 100) {
                int key = results.getInt(EMPNO_INDEX);
                EmployeeRecord emp = new EmployeeRecord(key);
                emp.setLastName(results.getString(LASTNAME_INDEX));
                emp.setFirstName(results.getString(FIRSTNAME_INDEX));
                emp.setDepartmentName(results.getString(DEPTNAME_INDEX));
                emp.updateSequenceNumber(results.getLong(SEQNO_INDEX));
                emp.setManagerNumber(results.getInt(MGRNO_INDEX));
                map.put(new Integer(key), emp);
                ++rows;
            }
            // Commit the transaction.
            session.commit();
            tranActive = false;
        } catch (Throwable t) {
            throw new LoaderException("preload failure: " + t, t);
        } finally {
            if (tranActive) {
                try {
                    session.rollback();
                } catch (Throwable t2) {
                    // Tolerate any rollback failures and
                    // allow original Throwable to be thrown.
                }
            }
            // Be sure to clean up other databases resources here
            // as well such a closing statements, result sets, etc.
        }
    }
```

This sample illustrates the following key points:

- The preloadMap backing map uses the Session object that is passed to it as the session argument.
- The Session.beginNoWriteThrough method is used to begin the transaction instead of the begin method.
- The Loader cannot be called for each put operation that occurs in this method for loading the map.
- The Loader can map columns of the employee table to a field in the EmployeeRecord Java object. The Loader catches all throwable exceptions that occur and throws a LoaderException exception with the caught throwable exception chained to it.
- The finally block ensures that any throwable exception that occurs between the time the beginNoWriteThrough method is called and the commit method is called cause the finally block to roll back the active transaction. This action is critical to ensure that any transaction that has been started by the preloadMap method is completed before returning to the caller. The finally block is a good place to perform other cleanup actions that might be needed, like closing the Java Database Connectivity (JDBC) connection and other JDBC objects.

The preloadMap sample is using a SQL select statement that selects all rows of the table. In your application-provided Loader, you might need to set one or more Loader properties to control how much of the table needs to be preloaded into the map.

Because the preloadMap method is only called one time during the BackingMap initialization, it is also a good place to run the one time Loader initialization code. Even if a Loader chooses not to prefetch data from the backend and load the data into the map, it probably needs to perform some other one time initialization to make other methods of the Loader more efficient. The following example illustrates caching the TransactionCallback object and OptimisticCallback object as instance variables of the Loader so that the other methods of the Loader do not have to make method calls to get access to these objects. This caching of the ObjectGrid plug-in values can be performed because after the BackingMap is initialized, the TransactionCallback and the OptimisticCallback objects cannot be changed or replaced. It is acceptable to cache these object references as instance variables of the Loader.

```
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.OptimisticCallback;
import com.ibm.websphere.objectgrid.plugins.TransactionCallback;

    // Loader instance variables.
    MyTransactionCallback ivTcb; // MyTransactionCallback

    // extends TransactionCallback
    MyOptimisticCallback ivOcb; // MyOptimisticCallback

    // implements OptimisticCallback
    // ...
    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
   [Replication programming]
        // Cache TransactionCallback and OptimisticCallback objects
        // in instance variables of this Loader.
        ivTcb = (MyTransactionCallback) session.getObjectGrid().getTransactionCallback();
        ivOcb = (MyOptimisticCallback) backingMap.getOptimisticCallback();
        // The remainder of preloadMap code (such as shown in prior example).
    }
```

For information about preloading and recoverable preloading as it pertains to replication failover, see the information about replication in the *Product Overview*.

Loaders with entity maps

If the loader is plugged into an entity map, the loader must handle tuple objects. Tuple objects are a special entity data format. The loader must conversion data between tuple and other data formats. For example, the get method returns a list of values that correspond to the set of keys that are passed in to the method. The passed-in keys are in the type of Tuple, says key tuples. Assuming that the loader persists the map with a database using JDBC, the get method must convert each key tuple into a list of attribute values that correspond to the primary key columns of the table that is mapped to the entity map, run the SELECT statement with the WHERE clause that uses converted attribute values as criteria to fetch data from database, and then convert the returned data into value tuples. The get method gets data from the database and converts the data into value tuples for passed-in key tuples, and then returns a list of value tuples correspond to the set of tuple keys that are passed in to the caller. The get method can perform one SELECT statement to fetch all data at one time, or run a SELECT statement for each key tuple. For programming details that show how to use the Loader when the data is store using an entity manager, see “Using a loader with entity maps and tuples” on page 168.

Related tasks

"Troubleshooting loaders" on page 274
Use this information to troubleshoot issues with your database loaders.

Related reference

"JPA loader programming considerations"
A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

JPA loader programming considerations

A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

eXtreme Scale entity and JPA entity

You can designate any POJO class as an eXtreme Scale entity using eXtreme Scale entity annotations, XML configuration, or both. You can also designate the same POJO class as a JPA entity using JPA entity annotations, XML configuration, or both.

eXtreme Scale entity: An eXtreme Scale entity represents persistent data that is stored in ObjectGrid maps. An entity object is transformed into a key tuple and a value tuple, which are then stored as key-value pairs in the maps. A tuple is an array of primitive attributes.

JPA entity: A JPA entity represents persistent data that is stored in a relational database automatically using container-managed persistence. The data is persisted in some form of a data storage system in the appropriate form, such as database tuples in a database.

When an eXtreme Scale entity is persisted, its relations are stored in other entity maps. For example, when you are persisting a Consumer entity with a one-to-many relation to a ShippingAddress entity, if cascade-persist is enabled, the ShippingAddress entity is stored in the shippingAddress map in tuple format. If you are persisting a JPA entity, the related JPA entities are also persisted to database tables if cascade-persist is enabled. When a POJO class is designated as both an eXtreme Scale entity and a JPA entity, the data can be persisted to both ObjectGrid entity maps and databases. Common uses follow:

- **Preload scenario**: An entity is loaded from a database using a JPA provider and persists it into ObjectGrid entity maps.
- **Loader scenario**: A Loader implementation is plugged in for the ObjectGrid entity maps so an entity stored in ObjectGrid entity maps can be persisted into or loaded from a database using JPA providers.

It is also common that a POJO class is designated as a JPA entity only. In that case, what is stored in the ObjectGrid maps are the POJO instances, versus the entity tuples in the ObjectGrid entity case.

Application design considerations for entity maps

When you are plugging in a JPALoader interface, the object instances are directly stored in the ObjectGrid maps.

However, you are when plugging in a JPAEntityLoader, the entity class is designated as both an eXtreme Scale entity and a JPA entity. In that case, treat this entity as if it has two persistent stores: the ObjectGrid entity maps and the JPA persistence store. The architecture becomes more complicated than the JPALoader case.

For more information about the JPAEntityLoader plug-in and application design considerations, see the information about the JPAEntityLoader plug-in in the *Administration Guide*. This information can also help if you plan to implement your own loader for the entity maps.

Performance considerations

Ensure that you set the proper eager or lazy fetch type for relationships. For example, a bidirectional one-to-many relationship Consumer to ShippingAddress, with OpenJPA to help explain the performance differences. In this example, a JPA query attempts `select o from Consumer o where . . .` to do a bulk load, and also load all related ShippingAddress objects. A one-to-many relationship defined in the Consumer class follows:

```
@Entity
public class Consumer implements Serializable {

    @OneToMany(mappedBy="consumer",cascade=CascadeType.ALL, fetch =FetchType.EAGER)
    ArrayList <ShippingAddress> addresses;
```

The many-to-one relation consumer defined in the ShippingAddress class follows:

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.EAGER)
    Consumer consumer;
}
```

If the fetch types of both relationships are configured as `eager`, OpenJPA uses N+1+1 queries to get all the Consumer objects and ShippingAddress objects, where N is the number of ShippingAddress objects. However if the ShippingAddress is changed to use lazy fetch type as follows, it only takes two queries to get all the data.

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.LAZY)
    Consumer consumer;
}
```

Although the query returns the same results, having a lower number of queries significantly decreases interaction with the database, which can increase application performance.

Related concepts

"Client-based JPA preload utility overview" on page 180
The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid.

Loaders overview

"Using a Loader" on page 157
With an eXtreme Scale Loader plug-in, an ObjectGrid map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or some other system. Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data, allowing hub-based caches to be built using ObjectGrid. A loader has the logic for reading and writing data to and from a persistent store.

"Writing a loader" on page 159
You can write your own loader plug-in implementation in your applications, which must follow the common WebSphere eXtreme Scale plug-in conventions.

Related tasks

"Troubleshooting loaders" on page 274
Use this information to troubleshoot issues with your database loaders.

JPAEntityLoader plug-in

The JPAEntityLoader plug-in is a built-in Loader implementation that uses Java Persistence API (JPA) to communicate with the database when you are using the EntityManager API. When you are using the ObjectMap API, use the JPALoader loader.

Loader details

Use the JPALoader plug-in when you are storing data using the ObjectMap API. Use the JPAEntityLoader plug-in when you are storing data using the EntityManager API.

Loaders provide two main functions:

1. **get**: In the get method, the JPAEntityLoader plug-in first calls the javax.persistence.EntityManager.find(Class entityClass, Object key) method to find the JPA entity. Then the plug-in projects this JPA entity into entity tuples. During the projection, both the tuple attributes and the association keys are stored in the value tuple. After processing each key, the get method returns a list of entity value tuples.
2. **batchUpdate**: The batchUpdate method takes a LogSequence object that contains a list of LogElement objects. Each LogElement object contains a key tuple and a value tuple. To interact with the JPA provider, you must first find the eXtreme Scale entity based on the key tuple. Based on the LogElement type, you run the following JPA calls:
   - **insert**: javax.persistence.EntityManager.persist(Object o)
   - **update**: javax.persistence.EntityManager.merge(Object o)
   - **remove**: javax.persistence.EntityManager.remove(Object o)

A LogElement with type **update** makes the JPAEntityLoader call the javax.persistence.EntityManager.merge(Object o) method to merge the entity. However, an **update** type LogElement might be the result of either a com.ibm.websphere.objectgrid.em.EntityManager.merge(object o) call or an attribute change of the eXtreme Scale EntityManager managed-instance. See the following example:

```
com.ibm.websphere.objectgrid.em.EntityManager em = og.getSession().getEntityManager();
em.getTransaction().begin();
Consumer c1 = (Consumer) em.find(Consumer.class, c.getConsumerId());
c1.setName("New Name");
em.getTransaction().commit();
```

In this example, an update type LogElement is sent to the JPAEntityLoader of the map consumer. The javax.persistence.EntityManager.merge(Object o) method is called to the JPA entity manager instead of an attribute update to the JPA-managed entity. Because of this changed behavior, some limitations exist with using this programming model.

Application design rules

Entities have relationships with other entities. Designing an application with relationships involved and with JPAEntityLoader plugged in requires additional considerations. The application should follow the following four rules, described in the following sections.

Limited relationship depth support

The JPAEntityLoader is only supported when using entities without any relationships or entities with single-level relationships. Relationships with more than one level, such as `Company > Department > Employee` are not supported.

One loader per map

Using the Consumer-ShippingAddress entity relationships as an example, when you load a consumer with eager fetch enabled, you could load all the related ShippingAddress objects. When you persist or merge a Consumer object, you could persist or merge related ShippingAddress objects if cascade-persist or cascade-merge is enabled.

You cannot plug in a loader for the root entity map which stores the Consumer entity tuples. You must configure a loader for each entity map.

Same cascade type for JPA and eXtreme Scale

Reconsider the scenario where the entity Consumer has a one-to-many relationship with ShippingAddress. You can look at the scenario where cascade-persist is enabled for this relationship. When a Consumer object is persisted into eXtreme Scale, the associated N number of ShippingAddress objects are also persisted into eXtreme Scale.

A persist call of the Consumer object with a cascade-persist relationship to ShippingAddress translates to one javax.persistence.EntityManager.persist(consumer) method call and *N* javax.persistence.EntityManager.persist(shippingAddress) method calls by the JPAEntityLoader layer. However, these N extra persist calls to ShippingAddress objects are unnecessary because of the cascade-persist setting from the JPA provider point of view. To solve this problem, eXtreme Scale provides a new method isCascaded on the LogElement interface. The isCascaded method indicates whether the LogElement is a result of an eXtreme Scale EntityManager cascade operation. In this example, the JPAEntityLoader of the ShippingAddress map receives *N* LogElement objects because of the cascade persist calls. The JPAEntityLoader finds out that the isCascaded method returns true and then ignores them without making any JPA calls. Therefore, from a JPA point of view, only one javax.persistence.EntityManager.persist(consumer) method call is received.

The same behavior is exhibited if you merge an entity or remove an entity with cascade enabled. The cascaded operations are ignored by the JPAEntityLoader plug-in.

The design of the cascade support is to replay the eXtreme Scale EntityManager operations to the JPA providers. These operations include persist, merge, and remove operations. To enable cascade support, verify that the cascade setting for the JPA and the eXtreme Scale EntityManager are the same.

Use entity update with caution

As previously described, the design of the cascade support is to replay eXtreme Scale EntityManager operations to the JPA providers. If your application calls the ogEM.persist(consumer) method to the eXtreme Scale EntityManager, even the associated ShippingAddress objects are persisted because of the cascade-persist setting, and the JPAEntityLoader only calls the jpAEM.persist(consumer) method to the JPA providers.

However, if your application updates a managed entity, this update translates to a JPA merge call by the JPAEntityLoader plug-in. In this scenario, support for multiple levels of relationships and key associations is not guaranteed. In this case, the best practice is to use the javax.persistence.EntityManager.merge(o) method instead of updating a managed entity.

Using a loader with entity maps and tuples

The entity manager converts all entity objects into tuple objects before they are stored in an WebSphere eXtreme Scale map. Every entity has a key tuple and a value tuple. This key-value pair is stored in the associated eXtreme Scale map for the entity. When you are using an eXtreme Scale map with a loader, the loader must interact with the tuple objects.

Overview

eXtreme Scale includes loader plug-ins that simplify integration with relational databases. The Java Persistence API (JPA) Loaders use a Java Persistence API to interact with the database and create the entity objects. The JPA loaders are compatible with eXtreme Scale entities.

Tuples

A tuple contains information about the attributes and associations of an entity. Primitive values are stored using their primitive wrappers. Other supported object types are stored in their native format. Associations to other entities are stored as a collection of key tuple objects that represent the keys of the target entities.

Each attribute or association is stored using a zero-based index. You can retrieve the index of each attribute using the getAttributePosition or getAssociationPosition methods. After the position is retrieved, it remains unchanged for the duration of the eXtreme Scale life cycle. The position can change when the eXtreme Scale is restarted. The setAttribute, setAssociation and setAssociations methods are used to update the elements in the tuple.

Attention: When you are creating or updating tuple objects, update every primitive field with a non-null value. Primitive values such as `int` should not be null. If you do not change the value to a default, poor performance issues can result, also affecting fields marked with the @Version annotation or version attribute in the entity descriptor XML file.

The following example further explains how to process tuples. For more information about defining entities for this example, see the information about the order entity schema, which is in the entity manager tutorial in the *Product Overview*.WebSphere eXtreme Scale is configured for using loaders with each of the entities. Additionally, only the Order entity is taken, and this specific entity has a many-to-one relationship with the Customer entity. The attribute name is `customer`, and it has a one-to-many relationship with the OrderLine entity.

Use the Projector to build Tuple objects automatically from entities. Using the Projector can simplify loaders when you are using an object-relational mapping utility such as Hibernate or JPA.

order.java

```
@Entity
public class Order
{
    @Id  String orderNumber;
    java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order") @OrderBy("lineNumber") List<OrderLine> lines;
}
```

customer.java

```
@Entity
public class Customer {
    @Id String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

orderLine.java

```
@Entity
public class OrderLine
{
    @Id @ManyToOne(cascade=CascadeType.PERSIST) Order order;
    @Id int lineNumber;
    @OneToOne(cascade=CascadeType.PERSIST) Item item;
    int quantity;
    double price;
}
```

A OrderLoader class that implements the Loader interface is shown in the following code. The following example assumes that an associated TransactionCallback plug-in is defined.

orderLoader.java

```
public class OrderLoader implements com.ibm.websphere.objectgrid.plugins.Loader {

 private EntityMetadata entityMetaData;
 public void batchUpdate(TxID txid, LogSequence sequence)
           throws LoaderException, OptimisticCollisionException {
        ...
        }
        public List get(TxID txid, List keyList, boolean forUpdate)
            throws LoaderException {
        ...
        }
        public void preloadMap(Session session, BackingMap backingMap)
            throws LoaderException {
  this.entityMetaData=backingMap.getEntityMetadata();
 }

}
```

The instance variable entityMetaData is initialized during the preLoadMap method call from the eXtreme Scale. The *entityMetaData* variable is not null if the Map is configured to use entities. Otherwise, the value is null.

The batchUpdate method

The batchUpdate method provides the ability to know what action the application intended to perform. Based on an insert, update or a delete operation, a connection can be opened to the database and the work performed. Because the key and values are of type Tuple, they must be transformed so the values make sense on the SQL statement.

The ORDER table was created with the following Data Definition Language (DDL) definition, as shown in the following code:

```
CREATE TABLE ORDER (ORDERNUMBER VARCHAR(250) NOT NULL, DATE TIMESTAMP, CUSTOMER_ID VARCHAR(250))
ALTER TABLE ORDER ADD CONSTRAINT PK_ORDER PRIMARY KEY (ORDERNUMBER)
```

The following code demonstrates how to convert a Tuple to an Object:

```
public void batchUpdate(TxID txid, LogSequence sequence)
            throws LoaderException, OptimisticCollisionException {
        Iterator iter = sequence.getPendingChanges();
        while (iter.hasNext()) {
            LogElement logElement = (LogElement) iter.next();
            Object key = logElement.getKey();
            Object value = logElement.getCurrentValue();

            switch (logElement.getType().getCode()) {
            case LogElement.CODE_INSERT:

1)              if (entityMetaData!=null) {

// The order has just one key orderNumber
2)                  String ORDERNUMBER=(String) getKeyAttribute("orderNumber", (Tuple) key);
// Get the value of date
3)                  java.util.Date unFormattedDate = (java.util.Date) getValueAttribute("date",(Tuple)value);
// The values are 2 associations. Lets process customer because
// the our table contains customer.id as primary key
4)                  Object[] keys= getForeignKeyForValueAssociation("customer","id",(Tuple) value);
                    //Order to Customer is M to 1. There can only be 1 key
5)                  String CUSTOMER_ID=(String)keys[0];
// parse variable unFormattedDate and format it for the database as formattedDate
6)                   String formattedDate = "2007-05-08-14.01.59.780272"; // formatted for DB2
// Finally, the following SQL statement to insert the record
7) //INSERT INTO ORDER (ORDERNUMBER, DATE, CUSTOMER_ID) VALUES(ORDERNUMBER,formattedDate, CUSTOMER_ID)
                }
                break;
            case LogElement.CODE_UPDATE:
                break;
            case LogElement.CODE_DELETE:
                break;
            }
        }

    }
// returns the value to attribute as stored in the key Tuple
 private Object getKeyAttribute(String attr, Tuple key) {
        //get key metadata
        TupleMetadata keyMD = entityMetaData.getKeyMetadata();
        //get position of the attribute
        int keyAt = keyMD.getAttributePosition(attr);
        if (keyAt > -1) {
            return key.getAttribute(keyAt);
        } else { // attribute undefined
            throw new IllegalArgumentException("Invalid position index for  "+attr);
        }

    }
// returns the value to attribute as stored in the value Tuple
    private Object getValueAttribute(String attr, Tuple value) {
        //similar to above, except we work with value metadata instead
        TupleMetadata valueMD = entityMetaData.getValueMetadata();
        int keyAt = valueMD.getAttributePosition(attr);
        if (keyAt > -1) {
            return value.getAttribute(keyAt);
        } else {
            throw new IllegalArgumentException("Invalid position index for  "+attr);
        }
    }
// returns an array of keys that refer to association.
    private Object[] getForeignKeyForValueAssociation(String attr, String fk_attr, Tuple value) {
        TupleMetadata valueMD = entityMetaData.getValueMetadata();
        Object[] ro;

        int customerAssociation = valueMD.getAssociationPosition(attr);
        TupleAssociation tupleAssociation = valueMD.getAssociation(customerAssociation);

        EntityMetadata targetEntityMetaData = tupleAssociation.getTargetEntityMetadata();

        Tuple[] customerKeyTuple = ((Tuple) value).getAssociations(customerAssociation);

        int numberOfKeys = customerKeyTuple.length;
        ro = new Object[numberOfKeys];

        TupleMetadata keyMD = targetEntityMetaData.getKeyMetadata();
        int keyAt = keyMD.getAttributePosition(fk_attr);
        if (keyAt < 0) {
            throw new IllegalArgumentException("Invalid position index for  " + attr);
        }
        for (int i = 0; i < numberOfKeys; ++i) {
            ro[i] = customerKeyTuple[i].getAttribute(keyAt);
        }

        return ro;

    }
```

1. Ensure that entityMetaData is not null, which implies that the key and value cache entries are of type Tuple. From the entityMetaData, Key TupleMetaData is retrieved, which really reflects only the key part of Order metadata.
2. Process the KeyTuple and get the value of Key Attribute orderNumber
3. Process the ValueTuple and get the value of attribute date
4. Process the ValueTuple and get the value of Keys from association customer
5. Extract CUSTOMER_ID. Based on relationship, an Order can only have one customer, we will only have one key. Hence the size of keys is 1. For simplicity, we skipped parsing of date to correct format.
6. Because this is an insert operation, the SQL statement is passed onto the data source connection to complete the insert operation.

Transaction demarcation and access to database is covered in “Writing a loader” on page 159.

The get method

If the key is not found in the cache, call the get method in the Loader plug-in to find the key.

The key is a Tuple. The first step is to convert the Tuple to primitive values that can be passed onto the SELECT SQL statement. After all the attributes are retrieved from the database, you must convert into Tuples. The following code demonstrates the Order class.

```
public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException {
  System.out.println("OrderLoader: Get called");
  ArrayList returnList = new ArrayList();

1)  if (entityMetaData != null) {
   int index=0;
   for (Iterator iter = keyList.iterator(); iter.hasNext();) {
2)    Tuple orderKeyTuple=(Tuple) iter.next();

    // The order has just one key orderNumber
3)    String ORDERNUMBERKEY = (String) getKeyAttribute("orderNumber",orderKeyTuple);
    //We need to run a query to get values of
4)    // SELECT CUSTOMER_ID, date FROM ORDER WHERE ORDERNUMBER='ORDERNUMBERKEY'

5)    //1) Foreign key: CUSTOMER_ID
```

```
6)    //2) date
    // Assuming those two are returned as
7)                              String  CUSTOMER_ID = "C001"; // Assuming Retrieved and initialized
8)    java.util.Date retrievedDate = new java.util.Date();
                                // Assuming this date reflects the one in database

    // We now need to convert this data into a tuple before returning

    //create a value tuple
9)    TupleMetadata valueMD = entityMetaData.getValueMetadata();
    Tuple valueTuple=valueMD.createTuple();


    //add retrievedDate object to Tuple
    int datePosition = valueMD.getAttributePosition("date");
10)    valueTuple.setAttribute(datePosition, retrievedDate);

    //Next need to add the Association
11)    int customerPosition=valueMD.getAssociationPosition("customer");
    TupleAssociation customerTupleAssociation =
                                valueMD.getAssociation(customerPosition);
    EntityMetadata customerEMD = customerTupleAssociation.getTargetEntityMetadata();
    TupleMetadata customerTupleMDForKEY=customerEMD.getKeyMetadata();
12)    int customerKeyAt=customerTupleMDForKEY.getAttributePosition("id");


    Tuple customerKeyTuple=customerTupleMDForKEY.createTuple();
    customerKeyTuple.setAttribute(customerKeyAt, CUSTOMER_ID);
13)    valueTuple.addAssociationKeys(customerPosition, new Tuple[] {customerKeyTuple});


14)    int linesPosition = valueMD.getAssociationPosition("lines");
    TupleAssociation linesTupleAssociation = valueMD.getAssociation(linesPosition);
    EntityMetadata orderLineEMD = linesTupleAssociation.getTargetEntityMetadata();
    TupleMetadata orderLineTupleMDForKEY = orderLineEMD.getKeyMetadata();
    int lineNumberAt = orderLineTupleMDForKEY.getAttributePosition("lineNumber");
    int orderAt = orderLineTupleMDForKEY.getAssociationPosition("order");

    if (lineNumberAt < 0 || orderAt < 0) {
     throw new IllegalArgumentException(
                                     "Invalid position index for lineNumber or order "+
                                     lineNumberAt + " " + orderAt);
    }
15) // SELECT LINENUMBER FROM ORDERLINE WHERE ORDERNUMBER='ORDERNUMBERKEY'
    // Assuming two rows of line number are returned with values 1 and 2

    Tuple orderLineKeyTuple1 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple1.setAttribute(lineNumberAt, new Integer(1));// set Key
    orderLineKeyTuple1.addAssociationKey(orderAt, orderKeyTuple);

    Tuple orderLineKeyTuple2 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple2.setAttribute(lineNumberAt, new Integer(2));// Init Key
    orderLineKeyTuple2.addAssociationKey(orderAt, orderKeyTuple);

16)    valueTuple.addAssociationKeys(linesPosition, new Tuple[]
                                  {orderLineKeyTuple1, orderLineKeyTuple2 });

    returnList.add(index, valueTuple);

    index++;

   }
  }else {
   // does not support tuples
  }
  return returnList;
 }
```

1. The get method is called when the ObjectGrid cache could not find the key and requests the loader to fetch. Test for entityMetaData value and proceed if not null.
2. The keyList contains Tuples.
3. Retrieve value of attribute orderNumber.
4. Run query to retrieve date (value) and customer ID (foreign key).
5. CUSTOMER_ID is a foreign key that must be set in the association tuple.
6. The date is a value and should already be set.
7. Since this example does not perform JDBC calls, CUSTOMER_ID is assumed.
8. Since this example does not perform JDBC calls, date is assumed.
9. Create value Tuple.
10. Set the value of date into the Tuple, based on its position.

11. Order has two associations. Start with the attribute customer which refers to the customer entity. You must have the value of ID to set in the Tuple.
12. Find the position of ID on the customer entity.
13. Set the values of the association keys only.
14. Also, lines is an association that must be set up as a group of association keys, in the same way as you did for customer association.
15. Since you must set up keys for the lineNumber associated with this order, run the SQL to retrieve lineNumber values.
16. Set up the association keys in the valueTuple. This completes the creation of the Tuple that is returned to the BackingMap.

This topic offers the steps create tuples, and a description of the Order entity only. Complete similar steps for the other entities, and the entire process that is tied together with the TransactionCallback plug-in. See “TransactionCallback plug-in” on page 148 for details.

Writing a loader with a replica preload controller

A Loader with a replica preload controller is a Loader that implements the ReplicaPreloadController interface in addition to the Loader interface.

Overview

The ReplicaPreloadController interface is designed to provide a way for a replica that becomes the primary shard to know whether the previous primary shard has completed the preload process. If the preload is partially completed, the information to pick up where the previous primary left is provided. With the ReplicaPreloadController interface implementation, a replica that becomes the primary continues the preload process where the previous primary left and continues to finish the overall preload.

In a distributed WebSphere eXtreme Scale environment, a map can have replicas and might preload large volume of data during initialization. The preload is a Loader activity and only occurs in the primary map during initialization. The preload might take a long time to complete if a large volume of data is preloaded. If the primary map has completed large portion of preload data, but is stopped for unknown reason during initialization, a replica becomes the primary. In this situation, the preloaded data that was completed by the previous primary is lost because the new primary normally performs an unconditional preload. With an unconditional preload, the new primary starts the preload process from the beginning and the previous preloaded data is ignored. If you want the new primary to pick up where the previous primary left during preload process, provide a Loader that implements the ReplicaPreloadController interface. For more information see the API documentation.

For information about Loaders, see the information about loaders in the *Administration Guide*. If you are interested in writing a regular Loader plug-in, see “Writing a loader” on page 159.

The ReplicaPreloadController interface has the following definition:

```
public interface ReplicaPreloadController
{
    public static final class Status
    {
        static public final Status PRELOADED_ALREADY = new Status(K_PRELOADED_ALREADY);
        static public final Status FULL_PRELOAD_NEEDED = new Status(K_FULL_PRELOAD_NEEDED);
        static public final Status PARTIAL_PRELOAD_NEEDED = new Status(K_PARTIAL_PRELOAD_NEEDED);
    }

    Status checkPreloadStatus(Session session, BackingMap bmap);
}
```

The following sections discuss some of the methods of the Loader and ReplicaPreloadController interface.

checkPreloadStatus method

When a Loader implements ReplicaPreloadController interface, the checkPreloadStatus method is called before the preloadMap method during map initialization. The return status of this method determines if the preloadMap method is called. If this method returns `Status#PRELOADED_ALREADY`, the preload method is not called. Otherwise, the preload method runs. Because of this behavior, this method should serve as the Loader initialization method. You must initialize the Loader properties in this method. This method must return the correct status, or the preload might not work as expected.

```
public Status checkPreloadStatus(Session session, BackingMap backingMap) {
        // When a loader implements ReplicaPreloadController interface, this method
    // will be called before preloadMap method during map initialization.
        // Whether the preloadMap method will be called depends on teh returned status of this method.
        // So, this method also serve as Loader's initialization method.
        // This method has to return the right staus, otherwise the preload may not work as expected.

        // Note: must initialize this loader instance here.
        ivOptimisticCallback = backingMap.getOptimisticCallback();
        ivBackingMapName = backingMap.getName();
        ivPartitionId = backingMap.getPartitionId();
        ivPartitionManager = backingMap.getPartitionManager();
        ivTransformer = backingMap.getObjectTransformer();
        preloadStatusKey = ivBackingMapName + "_" + ivPartitionId;

        try {
            // get the preloadStatusMap to retrieve preload status that could be set by other JVMs..
            ObjectMap preloadStatusMap = session.getMap(ivPreloadStatusMapName);

            // retrieve last recorded preload data chunk index.
            Integer lastPreloadedDataChunk = (Integer) preloadStatusMap.get(preloadStatusKey);

            if (lastPreloadedDataChunk == null) {
                preloadStatus = Status.FULL_PRELOAD_NEEDED;
            } else {
                preloadedLastDataChunkIndex = lastPreloadedDataChunk.intValue();
                if (preloadedLastDataChunkIndex == preloadCompleteMark) {
                    preloadStatus = Status.PRELOADED_ALREADY;
                } else {
                    preloadStatus = Status.PARTIAL_PRELOAD_NEEDED;
                }
            }

            System.out.println("TupleHeapCacheWithReplicaPreloadControllerLoader.checkPreloadStatus()
     -> map = " + ivBackingMapName + ", preloadStatusKey = " + preloadStatusKey
                    + ", retrieved lastPreloadedDataChunk =" + lastPreloadedDataChunk + ",
         determined preloadStatus = "
                    + getStatusString(preloadStatus));

        } catch (Throwable t) {
            t.printStackTrace();
        }

        return preloadStatus;
    }
```

preloadMap method

Running the preloadMap method depends on the returned result from checkPreloadStatus method. If the preloadMap method is called, it typically must retrieve preload status information from designated preload status map and determine how to proceed. Ideally, the preloadMap method should know if the preload is partially complete and exactly where to start. During the data preload, the preloadMap method should update the preload status on the designated preload status map. The preload status that is stored in the preload status map is retrieved by the checkPreloadStatus method when it needs to check the preload status.

```
public void preloadMap(Session session, BackingMap backingMap) throws LoaderException {
        EntityMetadata emd = backingMap.getEntityMetadata();
        if (emd != null && tupleHeapPreloadData != null) {
            // The getPreLoadData method is similar to fetching data from database.
            // These data will be push into cache as preload process.
            ivPreloadData = tupleHeapPreloadData.getPreLoadData(emd);

            ivOptimisticCallback = backingMap.getOptimisticCallback();
            ivBackingMapName = backingMap.getName();
            ivPartitionId = backingMap.getPartitionId();
            ivPartitionManager = backingMap.getPartitionManager();
            ivTransformer = backingMap.getObjectTransformer();
            Map preloadMap;

            if (ivPreloadData != null) {
                try {
                    ObjectMap map = session.getMap(ivBackingMapName);

                    // get the preloadStatusMap to record preload status.
                    ObjectMap preloadStatusMap = session.getMap(ivPreloadStatusMapName);

                    // Note: when this preloadMap method is invoked, the checkPreloadStatus
                    // has been called,
                    // Both preloadStatus and preloadedLastDataChunkIndex have been set.
                    // And the preloadStatus must be either PARTIAL_PRELOAD_NEEDED
         // or FULL_PRELOAD_NEEDED that
                    // will require a preload again.

                    // If large amount of data will be preloaded, the data usually is divided into
                    // few chunks and the preload process will process each chunk within its own tran.
                    // This sample only preload few entries and assuming each entry represent a chunk.
                    // so that the preload process an entry in a tran to simulate chunk preloading.

                    Set entrySet = ivPreloadData.entrySet();
                    preloadMap = new HashMap();
                    ivMap = preloadMap;

                    // The dataChunkIndex represent the data chunk that is in processing
                    int dataChunkIndex = -1;
                    boolean shouldRecordPreloadStatus = false;
                    int numberOfDataChunk = entrySet.size();
                    System.out.println("    numberOfDataChunk to be preloaded = "
           + numberOfDataChunk);

                    Iterator it = entrySet.iterator();
                    int whileCounter = 0;
                    while (it.hasNext()) {
                        whileCounter++;
                        System.out.println("preloadStatusKey = " + preloadStatusKey + " ,
            whileCounter = " + whileCounter);

                        dataChunkIndex++;

                        // if the current dataChunkIndex <= preloadedLastDataChunkIndex
                        // no need to process, becasue it has been preloaded by other JVM before.
                        // only need to process dataChunkIndex > preloadedLastDataChunkIndex
                        if (dataChunkIndex <= preloadedLastDataChunkIndex) {
                            System.out.println("ignore current dataChunkIndex = " + dataChunkIndex
                                    + " that has been previously preloaded.");
                            continue;
                        }

                        // Note: This sample simulate data chunk as an entry.
                        // each key represent a data chunk for simplicity.
                        // If the primary server or shard stopped for unknown reason, the preload status that
                        // indicates the progress of preload should be available in preloadStatusMap.
                        // A replica that become a primary can get the preload status and determine how to preload
                        // again.
                        // Note: recording preload status should be in the same tran as putting data into cache; so that
                        // if tran rollback or error, the recorded preload status is the actual status.
                        Map.Entry entry = (Entry) it.next();
                        Object key = entry.getKey();
                        Object value = entry.getValue();
                        boolean tranActive = false;

                        System.out.println("processing data chunk. map = " + this.ivBackingMapName
                                + ", current dataChunkIndex = " + dataChunkIndex + ", key = " + key);

                        try {
                            shouldRecordPreloadStatus = false; // re-set to false
                            session.beginNoWriteThrough();
                            tranActive = true;

                            if (ivPartitionManager.getNumOfPartitions() == 1) {
                                // if just only 1 partition, no need to deal with partition.
                                // just push data into cache
                                map.put(key, value);
                                preloadMap.put(key, value);
                                shouldRecordPreloadStatus = true;
                            } else if (ivPartitionManager.getPartition(key) == ivPartitionId) {
                                // if map is partitioned, need to consider the partition key
                                // only preload data that belongs to this partition.
                                map.put(key, value);
                                preloadMap.put(key, value);
                                shouldRecordPreloadStatus = true;
                            } else {
                                // ignore this entry, because it does not belong to this partition.
                            }

                            if (shouldRecordPreloadStatus) {
                                System.out.println("record preload status. map = " + this.ivBackingMapName
                                        + ", preloadStatusKey = " + preloadStatusKey + ", current dataChunkIndex ="
                                        + dataChunkIndex);
                                if (dataChunkIndex == numberOfDataChunk) {
                                    System.out.println("record preload status. map = " + this.ivBackingMapName
                                            + ", preloadStatusKey = " + preloadStatusKey + ", mark complete ="
                                            + preloadCompleteMark);
                                    // means we are at the lastest data chunk, if commit successfully, record preload
                                    // complete.
                                    // at this point, the preload is considered to be done
                                    // use -99 as special mark for preload complete status.
                                    preloadStatusMap.get(preloadStatusKey);
                                    // a put follow a get will become update if the get return an object,
                                    // otherwise, it will be insert.
                                    preloadStatusMap.put(preloadStatusKey, new Integer(preloadCompleteMark));

                                } else {
                                    // record preloaded current dataChunkIndex into preloadStatusMap
                                    // a put follow a get will become update if teh get return an object,
                                    // otherwise, it will be insert.
                                    preloadStatusMap.get(preloadStatusKey);
                                    preloadStatusMap.put(preloadStatusKey, new Integer(dataChunkIndex));
                                }
                            }

                            session.commit();
                            tranActive = false;

                            // to simulate preloading large amount of data
                            // put this thread into sleep for 30 secs.
                            // The real app should NOT put this thread to sleep
                            Thread.sleep(10000);

                        } catch (Throwable e) {
                            e.printStackTrace();
                            throw new LoaderException("preload failed with exception: " + e, e);
                        } finally {
                            if (tranActive && session != null) {
                                try {
                                    session.rollback();
                                } catch (Throwable e1) {
                                    // preload ignoring exception from rollback
                                }
                            }
                        }
                    }

                    // at this point, the preload is considered to be done for sure
                    // use -99 as special mark for preload complete status.
```

```
                // this is a insurance to make sure the complete mark is set.
                // besides, when partitioning, each partition does not know when is its last data chunk.
                // so the following block serves as the overall preload status complete reporting.
                System.out.println("Overall preload status complete -> record preload status. map = "
                        + this.ivBackingMapName + ", preloadStatusKey = " + preloadStatusKey + ", mark complete ="
                        + preloadCompleteMark);
                session.begin();
                preloadStatusMap.get(preloadStatusKey);
                // a put follow a get will become update if teh get return an object,
                // otherwise, it will be insert.
                preloadStatusMap.put(preloadStatusKey, new Integer(preloadCompleteMark));
                session.commit();

                ivMap = preloadMap;
            } catch (Throwable e) {
                e.printStackTrace();
                throw new LoaderException("preload failed with exception: " + e, e);
            }
        }
    }
}
```

Preload status map

You must use a preload status map to support the ReplicaPreloadController interface implementation. The preloadMap method should always check the preload status stored in the preload status map first and update the preload status in the preload status map whenever it pushes data into the cache. The checkPreloadStatus method can retrieve the preload status from preload status map, determine the preload status, and return the status to its caller. The preload status map should be in the same mapSet as other maps that have replica preload controller Loaders.

LogElement and LogSequence

When an application is making changes to a Map during a transaction, a LogSequence object tracks those changes. If the application changes an entry in the map, a corresponding LogElement object provides the details of the change.

Loaders are given a LogSequence object for a particular map whenever an application calls for a flush or commit to the transaction. The Loader iterates over the LogElement objects within the LogSequence object and applies each LogElement object to the backend.

ObjectGridEventListener listeners that are registered with an ObjectGrid also use LogSequence objects. These listeners are given a LogSequence object for each map in a committed transaction. Applications can use these listeners to wait for certain entries to change, like a trigger in a conventional database.

The following log-related interfaces or classes are provided by the eXtreme Scale framework:

- com.ibm.websphere.objectgrid.plugins.LogElement
- com.ibm.websphere.objectgrid.plugins.LogSequence
- com.ibm.websphere.objectgrid.plugins.LogSequenceFilter
- com.ibm.websphere.objectgrid.plugins.LogSequenceTransformer

LogElement interface

A LogElement represents an operation on an entry during a transaction. A LogElement object has several methods to get its various attributes. The most commonly used attributes are the type and the current value attributes fetched by getType() and getCurrentValue().

The type is represented by one of the constants defined in the LogElement interface: INSERT, UPDATE, DELETE, EVICT, FETCH, or TOUCH.

The current value represents the new value for the operation if it is INSERT, UPDATE or FETCH. If the operation is TOUCH, DELETE, or EVICT, then the current value is null. This value can be cast to ValueProxyInfo when a ValueInterface is in use.

See the API documentation for more details on the LogElement interface.

LogSequence interface

In most transactions, operations to more than one entry in a map occur, so multiple LogElement objects are created. You should create an object that behaves as a composite of multiple LogElement objects. The LogSequence interface serves this purpose by containing a list of LogElement objects.

See the API documentation for more details on the LogSequence interface.

Using LogElement and LogSequence

LogElement and LogSequence are widely used in eXtreme Scale and by ObjectGrid plug-ins that are written by users when operations are propagated from one component or server to another component or server. For example, a LogSequence object can be used by the distributed ObjectGrid transaction propagation function to propagate the changes to other servers, or it can be applied to the persistence store by the loader. LogSequence is mainly used by the following interfaces.

- com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener
- com.ibm.websphere.objectgrid.plugins.Loader
- com.ibm.websphere.objectgrid.plugins.Evictor
- com.ibm.websphere.objectgrid.Session

Loader example

This section demonstrates how the LogSequence and LogElement objects are used in a Loader. A Loader is used to load data from and persist data into a persistent store. The batchUpdate method of the Loader interface uses LogSequence object:

```
void batchUpdate(TxID txid, LogSequence sequence) throws
   LoaderException, OptimisticCollisionException;
```

The batchUpdate method is called when an ObjectGrid needs to apply all current changes to the Loader. The Loader is given a list of LogElement objects for the map, encapsulated in a LogSequence object. The implementation of the batchUpdate method must iterate over the changes and apply them to the backend. The following code snippet demonstrates how a Loader uses a LogSequence object. The snippet iterates over the set of changes and builds up three batch Java database connectivity (JDBC) statements: inserts, updates, and deletes:

```
public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException
{
    // Get a SQL connection to use.
    Connection conn = getConnection(tx);
    try
    {
    // Process the list of changes and build a set of prepared
    // statements for executing a batch update, insert, or delete
    // SQL operations. The statements are cached in stmtCache.
    Iterator iter = sequence.getPendingChanges();
    while ( iter.hasNext() )
    {
        LogElement logElement = (LogElement)iter.next();
        Object key = logElement.getCacheEntry().getKey();
        Object value = logElement.getCurrentValue();
        switch ( logElement.getType().getCode() )
        {
            case LogElement.CODE_INSERT:
                buildBatchSQLInsert( key, value, conn );
                break;
            case LogElement.CODE_UPDATE:
                buildBatchSQLUpdate( key, value, conn );
                break;
            case LogElement.CODE_DELETE:
                buildBatchSQLDelete( key, conn );
                break;
        }
    }
    // Run the batch statements that were built by above loop.
    Collection statements = getPreparedStatementCollection( tx, conn );
    iter = statements.iterator();
    while ( iter.hasNext() )
    {
        PreparedStatement pstmt = (PreparedStatement) iter.next();
        pstmt.executeBatch();
    }
} catch (SQLException e)
{
    LoaderException ex = new LoaderException(e);
    throw ex;
}
}
```

The previous sample illustrates the high-level logic of processing the LogSequence argument. However, the sample does not illustrate the details of how an SQL insert, update, or delete statement is built. The getPendingChanges method is called on the LogSequence argument to obtain an iterator of LogElement objects that a Loader needs to process, and the LogElement.getType().getCode() method is used to determine whether a LogElement is for an SQL insert, update, or delete operation.

Evictor sample

You can also use LogSequence and LogElement objects with an Evictor. An Evictor is used to evict the map entries from the backing map based on certain criteria. The apply method of the Evictor interface uses LogSequence.

```
/**
* This is called during cache commit to allow the evictor to track object usage
* in a backing map. This will also report any entries that have been successfully
* evicted.
*
* @param sequence LogSequence of changes to the map
*/
void apply(LogSequence sequence);
```

LogSequenceFilter and LogSequenceTransformer interfaces

Sometimes, it is necessary to filter the LogElement objects so that only LogElement objects with certain criteria are accepted, and reject other objects. For example, you might want to serialize a certain LogElement based on some criterion.

LogSequenceFilter solves this problem with the following method.

```
public boolean accept (LogElement logElement);
```

This method returns true if the given LogElement should be used in the operation, and returns false if the given LogElement should not be used.

LogSequenceTransformer is a class that uses the LogSequenceFilter function. It uses the LogSequenceFilter to filter out some LogElement objects and then serialize the accepted LogElement objects. This class has two methods. The first method follows.

```
public static void serialize(Collection logSequences, ObjectOutputStream stream,
   LogSequenceFilter filter, DistributionMode mode) throws IOException
```

This method allows the caller to provide a filter for determining which LogElements to include in the serialization process. The DistributionMode parameter allows the caller to control the serialization process. For example, if the distribution mode is invalidation only, then there is no need to serialize the value. The second method of this class is the inflate method, as follows.

```
public static Collection inflate(ObjectInputStream stream, ObjectGrid
   objectGrid) throws IOException, ClassNotFoundException
```

The inflate method reads the log sequence serialized form, which was created by the serialize method, from the provided object input stream.

Using eXtreme Scale with JPA

The Java Persistence API (JPA) is a specification that allows mapping Java objects to relational databases. JPA contains a full object-relational mapping (ORM) specification using Java language metadata annotations, XML descriptors, or both to define the mapping between Java objects and a relational database. A number of open-source and commercial implementations are available.

To use JPA, you must have a supported JPA provider, such as OpenJPA or Hibernate, JAR files, and a `META-INF/persistence.xml` file in your class path.

Client-based JPA preload utility overview

The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid.

This capability can simplify loading the eXtreme Scale maps when the queries to the database cannot be partitioned. A loader, such as a JPA Loader can also be used and is ideal when the data can be loaded in parallel.

The client-based JPA preload utility can use either the OpenJPA or Hibernate JPA implementations to load the ObjectGrid from a database. Because WebSphere eXtreme Scale does not directly interact with the database or Java Database Connectivity (JDBC), any database that OpenJPA or Hibernate supports can be used to load the ObjectGrid.

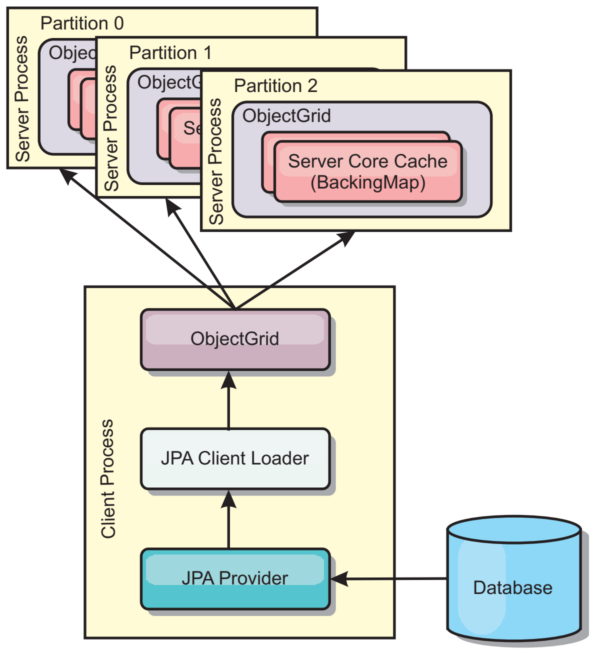

Figure 4. Client loader that uses JPA implementation to load the ObjectGrid

Typically, a user application provides a persistence unit name, an entity class name, and a JPA query to the client loader. The client loader retrieves the JPA entity manager based on the persistence unit name, uses the entity manager to query data from the database with the provided entity class and JPA query, and finally loads the data into the distributed ObjectGrid maps. When multi-level relations are involved in the query, can use a custom query string to optimize the performance. Optionally, a persistence property map could be provided to override the configured persistence properties.

A client loader can load data in two different modes, as displayed in the following table:

Table 11. Client loader modes

Mode	**Description**
Preload	Clears and loads all entries into the backing map. If the map is an entity map, any related entity maps will also be cleared if the ObjectGrid CascadeType.REMOVE option is enabled.
Reload	The JPA query is executed against the ObjectGrid to invalidate all the entities in the map that match the query. If the map is an entity map, any related entity maps will also be cleared if the ObjectGrid CascadeType.INVALIDATE option is enabled.

In either case, a JPA query is used to select and load the desired entities from the database and to store them in the ObjectGrid maps. If the ObjectGrid map is a non-entity map, the JPA entities will be detached and stored directly. If the

ObjectGrid map is an entity map, the JPA entities are stored as ObjectGrid entity tuples. You can provide a JPA query or use the default query `select o from EntityName o`.

For more information about configuring the client-based JPA preload utility, see the information in the *Programming Guide*

Related tasks

"Troubleshooting loaders" on page 274
Use this information to troubleshoot issues with your database loaders.

Related reference

"Client-based JPA preload utility programming"
The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid. You can implement preloading and reloading of data in your application.

"JPA loader programming considerations" on page 164
A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

Client-based JPA preload utility programming

The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid. You can implement preloading and reloading of data in your application.

Using the StateManager interface

Use the setObjectGridState method of the StateManager interface to set the ObjectGrid state to one of the following values: OFFLINE, ONLINE, QUIESCE or PRELOAD. The StateManager interface prevents other clients from accessing the ObjectGrid when it is not yet online.

For example, set the ObjectGrid state to PRELOAD before you load the maps with data. After the data load is complete, set the ObjectGrid state back to ONLINE. See the information about setting the availability of an ObjectGrid in the *Administration Guide* for more information.

When you are preloading different maps in one ObjectGrid, set the ObjectGrid state to `PRELOAD` one time and set the value back to `ONLINE` after all maps finish data loading. This coordination can be done by the ClientLoadCallback interface. Set the ObjectGrid state to `PRELOAD` after the first preStart notification from the ObjectGrid instance, and set it back to `ONLINE` after the last postFinish notification. If you need to coordinate between multiple Java virtual machines, your application must handle the coordination.

Client-based preload example

The flow of data pre-loading follows:

1. Clear the map to be preloaded. In the case of an entity map, if any relation is configured as cascade-remove, the related maps are also cleared.
2. Run the query to JPA for the entities in a batch. The batch size is 1000.
3. For each batch, build a key list and value list for each partition.
4. For each partition, call the data grid agent to insert or update the data on the server side directly if it is an eXtreme Scale client. If the grid is a local instance, directly insert or update the data in the ObjectGrid maps.

The following sample code snippet shows a simple client loading.

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
c.load(objectGrid, "CUSTOMER", "custPU", null,
    null, null, null, true, null);

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

In this example, the CUSTOMER map is configured as an entity map. The Customer entity class, which is configured in the ObjectGrid entity metadata descriptor XML file, has a one-to-many relation with Order entities. The Customer entity has the CascadeType.ALL option enabled on the relation to the Order entity.

Before the ClientLoader.load method is called, the ObjectGrid state is set to PRELOAD.

The parameters used in the ClientLoader.load method follow:

1. **objectGrid** : The ObjectGrid instance. It is a client-side ObjectGrid instance.
2. **"CUSTOMER"** : The map to be loaded. Since the Customer has a cascade-all relation with Order entities, the Order entities will be loaded too.
3. **"custPU"** : The JPA persistence unit name for the Customer and Order entities.
4. **null** : The persistenceProps map is null, which means the default persistence properties configured in the persistence.xml will be used.
5. **null** : The entityClass is configured as null. It will be set to the entity class configured in the ObjectGrid entity meta-data descriptor XML for the map "CUSTOMER", in this case, Customer.class.
6. **null** : The loadSql is null, which means the default "select o from CUSTOMER o" will be used to query the JPA entities.
7. **null** : The query parameter map is null.
8. **true** : This indicates the data loading mode is preload. Therefore, clear operations will be called to both the CUSTOMER and ORDER maps to clear all the data before loading due to the cascade-remove relation between them.
9. **null** : The ClientLoaderCallback is null.

For more information about the required parameters, see the ClientLoader API in the API documentation.

Reload example

Reloading a map is the same as preloading a map except that the isPreload argument is set to false in the ClientLoader.load method.

In the reload mode, the flow of data loading is as follows:

1. Run the provided query on the ObjectGrid map and invalidate all the results. In the case of an entity map, if any relation is configured with the CascadeType.INVALIDATE option, the related entities are also invalidated from their maps.

2. Run the provided query to the JPA to query the JPA entities in batch. The batch size is 1000.
3. For each batch, build a key list and value list for each partition.
4. For each partition, call the data grid agent to insert or update the data on the server side directly if it is an eXtreme Scale client. If the grid is a local eXtreme Scale configuration, directly insert or update the data in the ObjectGrid maps.

A reload sample follows:

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
String loadSql = "select c from CUSTOMER c
    where c.custId >= :startCustId and c.custId < :endCustId ";
Map<String, Long> params = new HashMap<String, Long>();
params.put("startCustId", 1000L);
params.put("endCustId", 2000L);

c.load(objectGrid, "CUSTOMER", "customerPU", null, null,
    loadSql, params, false, null);

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

Compared to the preload sample, the main difference is that a loadSql and parameters are provided. This sample only reloads the Customer data with id between 1000 and 2000.

Notice this query string observes both JPA query syntax and eXtreme Scale entity query syntax. This query string is important because it runs twice, once to ObjectGrid to invalidate the matched ObjectGrid entities and then to the JPA to load the matched JPA entities.

Calling a client loader in a Loader implementation

In the Loader interface, there is a preload method:

```
void preloadMap(Session session, BackingMap backingMap) throws
LoaderException;
```

This method signals the loader to preload the data into the map. A loader implementation can use a client loader to preload the data to all its partitions. For example, the JPA loader uses the client loader to preload data into the map.

For more information, see the JPA loaders overview topic in the *Product Overview*.

An example of how to preload the map using the client loader in the preloadMap method follows. The example first checks whether the current partition number is the same as the preload partition. If the partition number is not the same as the preload partition, no action occurs. If the partition numbers match, the client loader is called to load data into the maps. It is important to call the client loader in one and only one partition.

```
ObjectGrid og = session.getObjectGrid();
int partitionId = backingMap.getPartitionId();
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();

// Only call client loader data in one partition
if (partitionId == preloadPartition) {

    ClientLoader loader = ClientLoaderFactory.getClientLoader();

    // Call the client loader to load the data
    try {

        loader.load(og, backingMap.getName(), txCallback.getPersistenceUnitName(),
     null, entityClass, null, null, true, null);
    } catch (ObjectGridException e) {
        LoaderException le = new LoaderException("Exception caught in ObjectMap " +
     ogName + "." + mapName);
        le.initCause(e);
        throw le;
    }
}
```

Manual client load

The ClientLoader.load method provides a client load function that satisfies most scenarios. However, if you want to load data without the ClientLoader.load method, you can implement your own preload.

A template of a manual client load follows:

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

// Load the data
...

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

If you are loading the data from the client side, loading the data using a DataGrid agent could increase performance. By using a DataGrid agent, all the data reads and writes occur in the server process. You can also design your application to make sure DataGrid agents to multiple partitions run in parallel to further boost performance.

To implement data preload with a DataGrid agent, see the following example.

After you create the data preload implementation, you can create a generic Loader to complete the following tasks:

1. Query the data from database in batches.
2. Build a key list and value list for each partition.
3. For each partition, call the agentMgr.callReduceAgent(agent, aKey) method to run the agent in the server in a thread. By running in a thread, you can run agents concurrently on multiple partitions.

Example: Data preload with a DataGrid agent

If you are loading the data from the client side, loading the data using a DataGrid agent could increase performance. By using a DataGrid agent, all the data reads and writes occur in the server process. You can also design your application to make sure DataGrid agents to multiple partitions run in parallel to further boost performance.

An example of how to load the data with a DataGrid agent follows:

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

import com.ibm.websphere.objectgrid.NoActiveTransactionException;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TransactionException;
import com.ibm.websphere.objectgrid.datagrid.ReduceGridAgent;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class InsertAgent implements ReduceGridAgent, Externalizable {

    private static final long serialVersionUID = 6568906743945108310L;

    private List keys = null;

    private List vals = null;

    protected boolean isEntityMap;

    public InsertAgent() {
    }

    public InsertAgent(boolean entityMap) {
        isEntityMap = entityMap;
    }

    public Object reduce(Session sess, ObjectMap map) {
        throw new UnsupportedOperationException(
            "ReduceGridAgent.reduce(Session, ObjectMap)");
    }

    public Object reduce(Session sess, ObjectMap map, Collection arg2) {
        Session s = null;
        try {
            s = sess.getObjectGrid().getSession();
            ObjectMap m = s.getMap(map.getName());
            s.beginNoWriteThrough();
            Object ret = process(s, m);
            s.commit();
            return ret;
        } catch (ObjectGridRuntimeException e) {
            if (s.isTransactionActive()) {
                try {
                    s.rollback();
                } catch (TransactionException e1) {
                } catch (NoActiveTransactionException e1) {
                }
            }
            throw e;
        } catch (Throwable t) {
            if (s.isTransactionActive()) {
                try {
                    s.rollback();
                } catch (TransactionException e1) {
                } catch (NoActiveTransactionException e1) {
                }
            }
            throw new ObjectGridRuntimeException(t);
        }

    }

    public Object process(Session s, ObjectMap m) {
        try {

            if (!isEntityMap) {
                // In the POJO case, it is very straightforward,
                // we can just put everything in the
                // map using insert
                insert(m);
            } else {
                // 2. Entity case.
                // In the Entity case, we can persist the entities
                EntityManager em = s.getEntityManager();
                persistEntities(em);

            }
            return Boolean.TRUE;
        } catch (ObjectGridRuntimeException e) {
            throw e;
        } catch (ObjectGridException e) {
            throw new ObjectGridRuntimeException(e);
        } catch (Throwable t) {
            throw new ObjectGridRuntimeException(t);
        }

    }

    /**
     * Basically this is fresh load.
     * @param s
     * @param m
     * @throws ObjectGridException
     */
    protected void insert(ObjectMap m) throws ObjectGridException {

        int size = keys.size();

        for (int i = 0; i < size; i++) {
            m.insert(keys.get(i), vals.get(i));
        }

    }


    protected void persistEntities(EntityManager em) {
        Iterator<Object> iter = vals.iterator();

        while (iter.hasNext()) {
            Object value = iter.next();
            em.persist(value);
        }
    }
```

```
public Object reduceResults(Collection arg0) {
    return arg0;
}

public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    int v = in.readByte();
    isEntityMap = in.readBoolean();
    vals = readList(in);
    if (!isEntityMap) {
        keys = readList(in);
    }
}

public void writeExternal(ObjectOutput out) throws IOException {
    out.write(1);
    out.writeBoolean(isEntityMap);

    writeList(out, vals);
    if (!isEntityMap) {
        writeList(out, keys);
    }

}

public void setData(List ks, List vs) {
    vals = vs;
    if (!isEntityMap) {
        keys = ks;
    }
}

/**
 * @return Returns the isEntityMap.
 */
public boolean isEntityMap() {
    return isEntityMap;
}

static public void writeList(ObjectOutput oo, Collection l)
  throws IOException {
    int size = l == null ? -1 : l.size();
    oo.writeInt(size);
    if (size > 0) {
        Iterator iter = l.iterator();
        while (iter.hasNext()) {
            Object o = iter.next();
            oo.writeObject(o);
        }
    }
}

public static List readList(ObjectInput oi)
  throws IOException, ClassNotFoundException {
    int size = oi.readInt();
    if (size == -1) {
        return null;
    }


    ArrayList list = new ArrayList(size);
    for (int i = 0; i < size; ++i) {
        Object o = oi.readObject();
        list.add(o);
        }
        return list;
    }
}
```

Related concepts

"Client-based JPA preload utility overview" on page 180
The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid.

Related tasks

"Troubleshooting loaders" on page 274
Use this information to troubleshoot issues with your database loaders.

JPA time-based data updater

A Java Persistence API (JPA) time-based database updater updates the ObjectGrid maps with the latest changes in the database.

When changes are made directly to a database that is being fronted by WebSphere eXtreme Scale, those changes are not concurrently reflected in the eXtreme Scale grid. To properly implement eXtreme Scale as an in-memory database processing space, take into consideration that your grid can get out of sync with the database. Time-based database updater uses the System Change Number (SCN) capability in Oracle 10g and row change timestamp in DB2® 9.5 to monitor changes in the database for invalidation and update. The updater also allows applications to have a user-defined field for the same purpose.

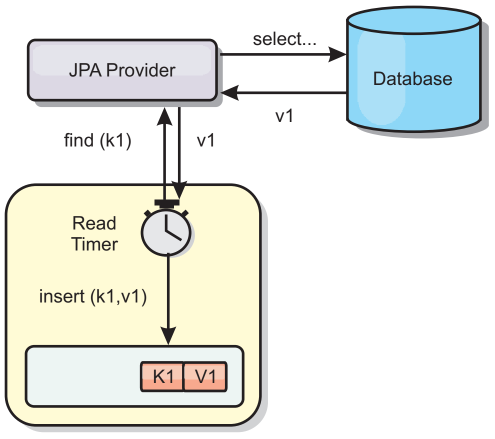

Figure 5. Periodic refresh

The time-based database updater periodically queries the database using JPA interfaces to get the JPA entities that represent the newly inserted and updated records in the database. To periodically update the records, every record in the database should have a timestamp to identify the time or sequence in which the record was last updated or inserted. It is not required that the timestamp be in a timestamp format. The timestamp value can be in an integer or long format, if it generates a unique, increasing value.

Several commercial databases have provided this capability.

For example, in DB2 9.5, you can define a column using the ROW CHANGE TIMESTAMP format as follows:

```
ROWCHGTS TIMESTAMP NOT NULL
     GENERATED ALWAYS
     FOR EACH ROW ON UPDATE AS
     ROW CHANGE TIMESTAMP
```

In Oracle, you can use the pseudo-column **ora_rowscn**, which represents the system change number of the record.

The time-based database updater updates the ObjectGrid maps in three different ways:

1. INVALIDATE_ONLY. Invalidate the entries in the ObjectGrid map if the corresponding records in the database have changed.
2. UPDATE_ONLY. Update the entries in the ObjectGrid map if the corresponding records in the database have changed. However, all the newly inserted records to the database are ignored.
3. INSERT_UPDATE. Update the existing entries in the ObjectGrid map with the latest values from the database. Also, all the newly inserted records to the database are inserted into the ObjectGrid map.

For more information about configuring the JPA time-based data updater, see the information in the *Administration Guide*.

Starting the JPA time-based updater

When you start the Java Persistence API (JPA) time-based updater, the ObjectGrid maps are updated with the latest changes in the database.

Before you begin

Configure the time-based updater. See the information about configuring a JPA time-based data updater in the *Administration Guide*.

About this task

For more information about how the Java Persistence API (JPA) time-based data updater works, see “JPA time-based data updater” on page 189.

Procedure

- Start a time-based database updater.
  - **Automatically for distributed eXtreme Scale:**

    If you create the timeBasedDBUpdate configuration for the backing map, the time-based database updater is automatically started when a distributed ObjectGrid primary shard is activated. For an ObjectGrid that has multiple partitions, the time-based database updater only starts at partition 0.
  - **Automatically for local eXtreme Scale:**

    If you create the timeBasedDBUpdate configuration for the backing map, the time-based database updater is automatically started when the local map is activated.
  - **Manually:**

    You can also manually start or stop a time-based database updater using the TimeBasedDBUpdater API.

    ```
    public synchronized void startDBUpdate(ObjectGrid objectGrid, String mapName,
     String punitName, Class entityClass, String timestampField, DBUpdateMode mode) {
    ```

    1. **ObjectGrid**: the ObjectGrid instance (local or client).

2. **mapName**: the name of the backing map for which the time-based database updater is started.
3. **punitName**: the JPA persistence unit name for creating a JPA entity manager factory; the default value is the name of the first persistence unit defined in the `persistence.xml` file.
4. **entityClass**: The entity class name used to interact with the Java Persistence API (JPA) provider; the entity class name is used to get JPA entities using entity queries.
5. **timestampField**: A timestamp field of the entity class to identify the time or sequence when a database back end record was last updated or inserted.
6. **mode**: The time-based database update mode; an INVALIDATE_ONLY type causes it to invalidate the entries in the ObjectGrid map if the corresponding records in the database have changed; an UPDATE_ONLY type indicates to update the existing entries in the ObjectGrid map with the latest values from the database; however, all the newly inserted records to the database are ignored; an INSERT_UPDATE type indicates to update the existing entries in the ObjectGrid map with the latest values from the database; also, all the newly inserted records to the database are inserted into the ObjectGrid map.

If you want to stop the time-based database updater, you can call the following method to stop the updater:

```
public synchronized void stopDBUpdate(ObjectGrid objectGrid, String mapName)
```

The ObjectGrid and mapName parameters should be the same as those passed in the startDBUpdate method.

- Create the timestamp field in your database.
  - **DB2**

    As a part of the optimistic locking feature, DB2 9.5 provides a row change timestamp feature. You can define a column ROWCHGTS using the ROW CHANGE TIMESTAMP format as follows:

```
ROWCHGTS TIMESTAMP NOT NULL
     GENERATED ALWAYS
     FOR EACH ROW ON UPDATE AS
     ROW CHANGE TIMESTAMP
```

    Then you can indicate the entity field which corresponds to this column as the timestamp field by either annotation or configuration. An example follows:

```
@Entity(name = "USER_DB2")
@Table(name = "USER1")
public class User_DB2 implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_DB2() {
    }

    public User_DB2(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = false, insertable = false)
    public Timestamp rowChgTs;
}
```

– **Oracle**

In Oracle, there is a pseudo-column `ora_rowscn` for the system change number of the record. You can use this column for the same purpose. An example of the entity that uses the ora_rowscn field as the time-based database update timestamp field follows:

```
@Entity(name = "USER_ORA")
@Table(name = "USER1")
public class User_ORA implements Serializable  {

    private static final long serialVersionUID = 1L;

    public User_ORA() {
    }

    public User_ORA(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ora_rowscn", updatable = false, insertable = false)
    public long rowChgTs;
}
```

– **Other databases**

For other types of databases, you can create a table column to track the changes. The column values have to be manually managed by the application that updates the table.

Take an Apache Derby database as an example: You can create a column "ROWCHGTS" to track the change numbers. Also, a latest change number is tracked for this table. Every time a record is inserted or updated, the latest change number for the table is incremented, and the ROWCHGTS column value for the record is updated with this incremented number.

```
@Entity(name = "USER_DER")
@Table(name = "USER1")
public class User_DER implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_DER() {
    }

    public User_DER(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = true, insertable = true)
    public long rowChgTs;
}
```

OptimisticCallback plug-in

Use the OptimisticCallback plug-in to customize versioning and comparison operations of cache objects when you are using the optimistic locking strategy.

You can provide a pluggable optimistic callback object that implements the com.ibm.websphere.objectgrid.plugins.OptimisticCallback interface. For entity maps, a high performance OptimisticCallback plug-in is automatically configured.

Purpose

Use the OptimisticCallback interface to provide optimistic comparison operations for the values of a map. An OptimisticCallback plug-in is required when you use the optimistic locking strategy. The product provides a default OptimisticCallback implementation. However, typically your application must plug in its own implementation of the OptimisticCallback interface.

Default implementation

The eXtreme Scale framework provides a default implementation of the OptimisticCallback interface that is used if the application does not plug in an application-provided OptimisticCallback object. The default implementation always returns the special value of NULL_OPTIMISTIC_VERSION as the version object for the value and never updates the version object. This action makes optimistic comparison a "no operation" function. In most cases, you do not want the "no operation" function to occur when you are using the optimistic locking strategy. Your applications must implement the OptimisticCallback interface and plug in their own OptimisticCallback implementations so that the default implementation is not used. However, at least one scenario exists where the default provided OptimisticCallback implementation is useful. Consider the following situation:

- A loader is plugged in for the backing map.
- The loader knows how to perform the optimistic comparison without assistance from an OptimisticCallback plug-in.

How can the loader perform optimistic versioning without assistance from an OptimisticCallback object? The loader has knowledge of the value class object and knows which field of the value object is used as an optimistic versioning value. For example, suppose the following interface is used for the value object for the employees map:

```
public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}
```

In this example, the loader knows that it can use the getSequenceNumber method to get the current version information for an Employee value object. The loader increments the returned value to generate a new version number before it updates the persistent storage with the new Employee value. For a Java database connectivity (JDBC) loader, the current sequence number in the WHERE clause of an overqualified SQL UPDATE statement is used, and it uses the new generated sequence number to set the sequence number column to the new sequence number value. Another possibility is that the loader makes use of some backend-provided function that automatically updates a hidden column that can be used for optimistic versioning.

In some situations, a stored procedure or trigger can possibly be used to help maintain a column that holds versioning information. If the loader is using one of these techniques for maintaining optimistic versioning information, then the application does not need to provide an OptimisticCallback implementation. The default OptimisticCallback implementation is usable in this scenario because the loader can handle optimistic versioning without any assistance from an OptimisticCallback object.

Default implementation for entities

Entities are stored in the ObjectGrid using tuple objects. The default OptimisticCallback implementation behavior is similar to the behavior for non-entity maps. However, the version field in the entity is identified using the @Version annotation or the version attribute in the entity descriptor XML file.

The version attribute can be of the following types: int, Integer, short, Short, long, Long or java.sql.Timestamp. An entity must only have one version attribute defined. Only set the version attribute during construction. After the entity is persisted, the value of the version attribute should not be modified.

If a version attribute is not configured and the optimistic locking strategy is used, then the entire tuple is implicitly versioned using the entire state of the tuple, which is much more expensive

In the following example, the Employee entity has a long version attribute named SequenceNumber:

```
@Entity
public class Employee
{
private long sequence;
    // Sequential sequence number used for optimistic versioning.
    @Version
    public long getSequenceNumber() {
        return sequence;
    }
    public void setSequenceNumber(long newSequenceNumber) {
        this.sequence = newSequenceNumber;
    }
    // Other get/set methods for other fields of Employee object.
}
```

Writing an OptimisticCallback plug-in

An OptimisticCallback plug-in must to implement the OptimisticCallback interface and follow the common ObjectGrid plug-in conventions. See the OptimisticCallback interface in the API documentation for more information.

The following list provides a description or consideration for each of the methods in the OptimisticCallback interface:

NULL_OPTIMISTIC_VERSION

This special value is returned by the getVersionedObjectForValue method if the OptimisticCallback implementation does not require version checking. The built-in plugin implementation of the com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback class uses this value because versioning is disabled when you are specifying this plug-in implementation.

getVersionedObjectForValue method

The getVersionedObjectForValue method might return a copy of the value or an attribute of the value that can be used for versioning purposes. This method is called whenever an object is associated with a transaction. When no Loader is plugged into a backing map, the backing map uses this value at commit time to perform an optimistic version comparison. The optimistic version comparison is used by the backing map to ensure that the version has not changed after this transaction first accessed the map entry that was modified by this transaction. If another transaction had already modified the version for this map entry, the version comparison fails and the backing map displays an OptimisticCollisionException exception to force the transaction to roll back. If a Loader is plugged in, the backing map does not use the optimistic versioning information. Instead, the Loader is responsible for performing the optimistic versioning comparison and updating the versioning information when necessary. The Loader typically gets the initial versioning object from the LogElement passed to the batchUpdate method on the loader, which is called when a flush operation occurs or a transaction is committed.

The following code shows the implementation used by the EmployeeOptimisticCallbackImpl object:

```
public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}
```

As demonstrated in the previous example, the sequenceNumber attribute is returned in a java.lang.Long object as expected by the Loader, which implies that the same person that wrote the Loader either wrote the EmployeeOptimisticCallbackImpl implementation or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl - for example, agreed on the value returned by the getVersionedObjectForValue method. The default OptimisticCallback plug-in returns the special value NULL_OPTIMISTIC_VERSION as the version object.

updateVersionedObjectForValue method

This method is called whenever a transaction has updated a value and a new versioned object is needed. If the getVersionedObjectForValue method returns an attribute of the value, this method typically updates the attribute value with a new version object. If getVersionedObjectForValue method returns a copy of the value, this method typically does not complete any actions. The default OptimisticCallback plug-in does not complete any actions with this method because the default implementation of getVersionedObjectForValue always returns the special value NULL_OPTIMISTIC_VERSION as the version object. The following example shows the implementation used by the EmployeeOptimisticCallbackImpl object that is used in the OptimisticCallback section:

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

As demonstrated in the previous example, the sequenceNumber attribute increments by one so that the next time the getVersionedObjectForValue method is called, the java.lang.Long value that is returned has a long value that is the original sequence number value plus one, for example, is the next version value for this employee instance. This example implies that the same person that wrote the Loader either wrote EmployeeOptimisticCallbackImpl or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl.

serializeVersionedValue method

This method writes the versioned value to the specified stream. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the appropriate serialization. The default implementation calls the writeObject method.

inflateVersionedValue method

This method takes the serialized version of the versioned value and returns the actual versioned value object. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the appropriate deserialization. The default implementation calls the readObject method.

Using application-provided OptimisticCallback object

You have two approaches to add an application-provided OptimisticCallback object into the BackingMap configuration: XML configuration and programmatic configuration.

XML configuration approach to plug in an OptimisticCallback object

The application can use an XML file to plug in its OptimisticCallback object as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
    <objectGrid name="grid1">
        <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
    </objectGrid>
</objectGrids>

<backingMapPluginCollections>
    <backingMapPluginCollection id="employees">
        <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

Programmatically plug in an OptimisticCallback object

The following example demonstrates how an application can programmatically plug in an OptimisticCallback object for the employee backing map in the local grid1 ObjectGrid instance:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

ObjectTransformer plug-in

With the ObjectTransformer plug-in, you can serialize, deserialize, and copy objects in the cache for increased performance.

If you see performance issues with processor usage, add an ObjectTransformer plug-in to each map. If you do not provide an ObjectTransformer plug-in, up to 60-70 percent of the total processor time is spent serializing and copying entries.

Purpose

With the ObjectTransformer plug-in, your applications can provide custom methods for the following operations:
- Serialize or deserialize the key for an entry
- Serialize or deserialize the value for an entry
- Copy a key or value for an entry

If no ObjectTransformer plug-in is provided, you must be able to serialize the keys and values because the ObjectGrid uses a serialize and deserialize sequence to copy the objects. This method is expensive, so use an ObjectTransformer plug-in when performance is critical. The copying occurs when an application looks up an object in a transaction for the first time. You can avoid this copying by setting the copy mode of the Map to NO_COPY or reduce the copying by setting the copy mode to COPY_ON_READ. Optimize the copy operation when needed by the application by providing a custom copy method on this plug-in. Such a plug-in can reduce the copy overhead from 65–70 percent to 2/3 percent of total processor time.

The default copyKey and copyValue method implementations first attempt to use the clone method, if the method is provided. If no clone method implementation is provided, the implementation defaults to serialization.

Object serialization is also used directly when the eXtreme Scale is running in distributed mode. The LogSequence uses the ObjectTransformer plug-in to help serialize keys and values before transmitting the changes to peers in the ObjectGrid. You must take care when providing a custom serialization method instead of using the built-in Java developer kit serialization. Object versioning is a complex issue and you might encounter problems with version compatibility if you do not ensure that your custom methods are designed for versioning.

The following list describes how the eXtreme Scale tries to serialize both keys and values:

- If a custom ObjectTransformer plug-in is written and plugged in, eXtreme Scale calls methods in the ObjectTransformer interface to serialize keys and values and get copies of object keys and values.
- If a custom ObjectTransformer plug-in is not used, eXtreme Scale serializes and deserializes values according to the default. If the default plug-in is used, each object is implemented as externalizable or is implemented as serializable.
  - If the object supports the Externalizable interface, the writeExternal method is called. Objects that are implemented as externalizable lead to better performance.
  - If the object does not support the Externalizable interface and does implement the Serializable interface,, the object is saved using the ObjectOutputStream method.

Using the ObjectTransformer interface

An ObjectTransformer object must implement the ObjectTransformer interface and follow the common ObjectGrid plug-in conventions.

Two approaches, programmatic configuration and XML configuration, are used to add an ObjectTransformer object into the BackingMap configuration as follows.

XML configuration approach to plug in an ObjectTransformer

Assume that the class name of the ObjectTransformer implementation is the com.company.org.MyObjectTransformer class. This class implements the ObjectTransformer interface. An ObjectTransformer implementation can be configured using the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
     <backingMap name="myMap" pluginCollectionRef="myMap" />
```

```
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
 <backingMapPluginCollection id="myMap">
      <bean id="ObjectTransformer" className="com.company.org.MyObjectTransformer" />
 </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

Programmatically plug in an ObjectTransformer object

The following code snippet creates the custom ObjectTransformer object and adds it to a BackingMap:

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap backingMap = myGrid.getMap("myMap");
MyObjectTransformer myObjectTransformer = new MyObjectTransformer();
backingMap.setObjectTransformer(myObjectTransformer);
```

ObjectTransformer usage scenarios

You can use the ObjectTransformer plug-in in the following situations:

- Non-serializable object
- Serializable object but improve serialization performance
- Key or value copy

In the following example, ObjectGrid is used to store the Stock class:

```
/**
* Stock object for ObjectGrid demo
*
*
*/
public class Stock implements Cloneable {
    String ticket;
    double price;
    String company;
    String description;
    int serialNumber;
    long lastTransactionTime;
    /**
    * @return Returns the description.
    */
    public String getDescription() {
        return description;
    }
    /**
    * @param description The description to set.
    */
    public void setDescription(String description) {
        this.description = description;
    }
    /**
    * @return Returns the lastTransactionTime.
    */
    public long getLastTransactionTime() {
        return lastTransactionTime;
    }
    /**
    * @param lastTransactionTime The lastTransactionTime to set.
    */
    public void setLastTransactionTime(long lastTransactionTime) {
        this.lastTransactionTime = lastTransactionTime;
    }
    /**
    * @return Returns the price.
    */
    public double getPrice() {
        return price;
    }
    /**
    * @param price The price to set.
    */
    public void setPrice(double price) {
        this.price = price;
    }
    /**
    * @return Returns the serialNumber.
    */
    public int getSerialNumber() {
        return serialNumber;
    }
    /**
    * @param serialNumber The serialNumber to set.
    */
    public void setSerialNumber(int serialNumber) {
        this.serialNumber = serialNumber;
    }
    /**
    * @return Returns the ticket.
    */
    public String getTicket() {
        return ticket;
    }
    /**
    * @param ticket The ticket to set.
    */
    public void setTicket(String ticket) {
        this.ticket = ticket;
    }
    /**
    * @return Returns the company.
    */
    public String getCompany() {
        return company;
    }
    /**
    * @param company The company to set.
    */
    public void setCompany(String company) {
        this.company = company;
    }
    //clone
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

You can write a custom object transformer class for the Stock class:

```
/**
* Custom implementation of ObjectGrid ObjectTransformer for stock object
*
*/
public class MyStockObjectTransformer implements ObjectTransformer {
/* (non-Javadoc)
* @see
* com.ibm.websphere.objectgrid.plugins.ObjectTransformer#serializeKey
* (java.lang.Object,
* java.io.ObjectOutputStream)
*/
public void serializeKey(Object key, ObjectOutputStream stream) throws IOException {
    String ticket= (String) key;
    stream.writeUTF(ticket);
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#serializeValue(java.lang.Object,
java.io.ObjectOutputStream)
*/
public void serializeValue(Object value, ObjectOutputStream stream) throws IOException {
    Stock stock= (Stock) value;
    stream.writeUTF(stock.getTicket());
    stream.writeUTF(stock.getCompany());
    stream.writeUTF(stock.getDescription());
    stream.writeDouble(stock.getPrice());
    stream.writeLong(stock.getLastTransactionTime());
    stream.writeInt(stock.getSerialNumber());
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateKey(java.io.ObjectInputStream)
*/
public Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException {
    String ticket=stream.readUTF();
    return ticket;
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateValue(java.io.ObjectInputStream)
*/

public Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException {
    Stock stock=new Stock();
    stock.setTicket(stream.readUTF());
    stock.setCompany(stream.readUTF());
    stock.setDescription(stream.readUTF());
    stock.setPrice(stream.readDouble());
    stock.setLastTransactionTime(stream.readLong());
    stock.setSerialNumber(stream.readInt());
    return stock;
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyValue(java.lang.Object)
*/
public Object copyValue(Object value) {
    Stock stock = (Stock) value;
    try {
        return stock.clone();
    }
    catch (CloneNotSupportedException e)
    {
        // display exception message    }
}

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyKey(java.lang.Object)
*/
public Object copyKey(Object key) {
    String ticket=(String) key;
    String ticketCopy= new String (ticket);
    return ticketCopy;
}
}
```

Then, plug in this custom MyStockObjectTransformer class into the BackingMap:

```
ObjectGridManager ogf=ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogf.getObjectGrid("NYSE");
BackingMap bm = og.defineMap("NYSEStocks");
MyStockObjectTransformer ot = new MyStockObjectTransformer();
bm.setObjectTransformer(ot);
```

WebSphereTransactionCallback plug-in

When you use the WebSphereTransactionCallback plug-in, enterprise applications that are running in a WebSphere Application Server environment can manage ObjectGrid transactions.

When you are using an ObjectGrid session within a method that is configured to use container-managed transactions, the enterprise container automatically begins, commits or rolls back the ObjectGrid transaction. When you are using Java Transaction API (JTA) UserTransaction objects, the ObjectGrid transaction is managed by the UserTransaction object automatically.

For a detailed discussion of the implementation of this plug-in, see “External transaction managers” on page 155

Note: The ObjectGrid does not support 2-phase, XA transactions. This plug-in does not enlist the ObjectGrid transaction with the transaction manager. Therefore, if the ObjectGrid fails to commit, any other resources that are managed by the XA transaction do not roll back.

Enabling the WebSphereTransactionCallback plug-in

You can enable the WebSphereTransactionCallback into the ObjectGrid configuration with programmatic configuration or XML configuration.

XML configuration approach to plug in the WebSphereTransactionCallback object

The following XML configuration creates the WebSphereTransactionCallback object and adds it to an ObjectGrid. The following text must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="myGrid">
            <bean id="TransactionCallback" className=
 "com.ibm.websphere.objectgrid.plugins.builtins.WebSphereTransactionCallback" />

        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

Programmatically plug in the WebSphereTransactionCallback object

The following code snippet creates the WebSphereTransactionCallback object and adds it to an ObjectGrid:

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
WebSphereTransactionCallback wsTxCallback= new WebSphereTransactionCallback ();
myGrid.setTransactionCallback(wsTxCallback);
```

Chapter 5. Administration APIs

In addition to system application programming interfaces (APIs), WebSphere eXtreme Scale also includes administration APIs that allow applications to monitor and administer servers and clients.

Embedded server API

WebSphere eXtreme Scale includes application programming interfaces (APIs) and system programming interfaces for embedding eXtreme Scale servers and clients within your existing Java applications. The following topic describes the available embedded server APIs.

Instantiating the eXtreme Scale server

You can use several properties to configure the eXtreme Scale server instance, which you can retrieve from the ServerFactory.getServerProperties method. The ServerProperties object is a singleton, so each call to the getServerProperties method retrieves the same instance.

You can create a new server with the following code.

```
Server server = ServerFactory.getInstance();
```

All properties set before the first invocation of getInstance are used to initialize the server.

Setting server properties

You can set the server properties until the ServerFactory.getInstance is called for the first time. The first call of the getInstance method instantiates the eXtreme Scale server, and reads all the configured properties. Setting the properties after creation has no effect. the following example shows how to set properties prior to instantiating a Server instance.

```
// Get the server properties associated with this process.
ServerProperties serverProperties = ServerFactory.getServerProperties();

// Set the server name for this process.
serverProperties.setServerName("EmbeddedServerA");

// Set the name of the zone this process is contained in.
serverProperties.setZoneName("EmbeddedZone1");

// Set the end point information required to bootstrap to the catalog service.
serverProperties.setCatalogServiceBootstrap("localhost:2809");

// Set the ORB listener host name to use to bind to.
serverProperties.setListenerHost("host.local.domain");

// Set the ORB listener port to use to bind to.
serverProperties.setListenerPort(9010);

// Turn off all MBeans for this process.
serverProperties.setMBeansEnabled(false);

Server server = ServerFactory.getInstance();
```

Embedding the catalog service

Any JVM setting that is flagged by the CatalogServerProperties.setCatalogServer method can host the catalog service for eXtreme Scale. This method indicates to the eXtreme Scale server runtime to instantiate the catalog service when the server is started. The following code shows how to instantiate the eXtreme Scale catalog server:

```
CatalogServerProperties catalogServerProperties =
 ServerFactory.getCatalogProperties();
catalogServerProperties.setCatalogServer(true);

Server server = ServerFactory.getInstance();
```

Embedding the eXtreme Scale container

Issue the Server.createContainer method for any JVM to host multiple eXtreme Scale containers. The following code shows how to instantiate an eXtreme Scale container:

```
Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURI().toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURI().toURL());
Container container = server.createContainer(policy);
```

Self-contained server process

You can start all the services together, which is useful for development and also practical in production. By starting the services together, a single process does all of the following: Starts the catalog service, starts a set of containers, and runs the client connection logic. Starting the services in this way sorts out programming issues prior to deploying in a distributed environment. The following code shows how to instantiate a self-contained eXtreme Scale server:

```
CatalogServerProperties catalogServerProperties =
 ServerFactory.getCatalogProperties();
catalogServerProperties.setCatalogServer(true);

Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURI().toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURI().toURL());
Container container = server.createContainer(policy);
```

Embedding eXtreme Scale in WebSphere Application Server

The configuration for eXtreme Scale is set up automatically when you install WebSphere Extended Deployment DataGrid in a WebSphere Application Server environment. You are not required to set any properties before you access the server to create a container. The following code shows how to instantiate an eXtreme Scale server in WebSphere Application Server:

```
Server server = ServerFactory.getInstance();
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy(
    new File("META-INF/embeddedDeploymentPolicy.xml").toURI().toURL(),
    new File("META-INF/embeddedObjectGrid.xml").toURI().toURL);
Container container = server.createContainer(policy);
```

For a step by step example on how to start an embedded catalog service and container programmatically, see "Using the embedded server API" on page 205.

Using the embedded server API

With WebSphere eXtreme Scale, you can use a programmatic API for managing the life cycle of embedded servers and containers. You can programmatically configure the server with any of the options that you can also configure with the command line options or file-based server properties. You can configure the embedded server to be a container server, a catalog service, or both.

Before you begin

You must have a method for running code from within an already existing Java virtual machine. The eXtreme Scale classes must be available through the class loader tree.

About this task

You can run many administration tasks with the Administration API. One common use of the API is as an internal server for storing Web application state. The Web server can start an embedded WebSphere eXtreme Scale server, report the container server to the catalog service, and the server is then added as a member of a larger distributed grid. This usage can provide scalability and high availability to an otherwise volatile data store.

You can programmatically control the complete life cycle of an embedded eXtreme Scale server. The examples are as generic as possible and only show direct code examples for the outlined steps.

Procedure

1. Obtain the ServerProperties object from the ServerFactory class and configure any necessary options.

   Every eXtreme Scale server has a set of configurable properties. When a server starts from the command line, those properties are set to defaults, but you can override several properties by providing an external source or file. In the embedded scope, you can directly set the properties with a ServerProperties object. You must set these properties before you obtain a server instance from the ServerFactory class. The following example snippet obtains a ServerProperties object, sets the CatalogServiceBootStrap field, and initializes several optional server settings. See the API documentation for a list of the configurable settings.

   ```
   ServerProperties props = ServerFactory.getServerProperties();
   props.setCatalogServiceBootstrap("host:port"); // required to connect to specific catalog service
   props.setServerName("ServerOne"); // name server
   props.setTraceSpecification("com.ibm.ws.objectgrid=all=enabled"); // Sets trace spec
   ```

2. If you want the server to be a catalog service, obtain the CatalogServerProperties object.

   Every embedded server can be a catalog service, a container server, or both a container server and a catalog service. The following example obtains the CatalogServerProperties object, enables the catalog service option, and configures various catalog service settings.

   ```
   CatalogServerProperties catalogProps = ServerFactory.getCatalogProperties();
   catalogProps.setCatalogServer(true); // false by default, it is required to set as
                  // a catalog service
   catalogProps.setQuorum(true); // enables / disables quorum
   ```

3. Obtain a Server instance from the ServerFactory class. The Server instance is a process-scoped singleton that is responsible for managing the membership in the grid. After this instance has been instantiated, this process is connected

and is highly available with the other servers in the grid. The following example shows how to create the Server instance:

```
 Server server = ServerFactory.getInstance();
```

Reviewing the previous example, the ServerFactory class provides a static method that returns a Server instance. The ServerFactory class is intended to be the only interface for obtaining a Server instance. Therefore, the class ensures that the instance is a singleton, or one instance for each JVM or isolated classloader. The getInstance method initializes the Server instance. You must configure all the server properties before you initialize the instance. The Server class is responsible for creating new Container instances. You can use both the ServerFactory and Server classes for managing the life cycle of the embedded Server instance.

4. Start a Container instance using the Server instance.

   Before shards can be placed on an embedded server, you must create a container on the server. The Server interface has a createContainer method that takes a DeploymentPolicy argument. The following example uses the server instance that you obtained to create a container using a created DeploymentPolicy file. Note that Containers require a classloader that has the application binaries available to it for serialization. You can make these binaries available by calling the createContainer method with the Thread context classloader set to the classloader that you want to use.

```
DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy
(new URL("file://urltodeployment.xml"),
 new URL("file://urltoobjectgrid.xml"));
Container container = server.createContainer(policy);
```

5. Remove and clean up a container.

   You can remove and clean up a container server by using the running the teardown method on the obtained Container instance. Running the teardown method on a container properly cleans up the container and removes the container from the embedded server.

   The process of cleaning up the container includes the movement and tearing down of all the shards that are placed within that container. Each server can contain many containers and shards. Cleaning up a container does not affect the life cycle of the parent Server instance. The following example demonstrates how to run the teardown method on a server. The teardown method is made available through the ContainerMBean interface. By using the ContainerMBean interface, if you no longer have programmatic access to this container, you can still remove and clean up the container with its MBean. A terminate method also exists on the Container interface, do not use this method unless it is absolutely needed. This method is more forceful and does not coordinate appropriate shard movement and clean up.

```
container.teardown();
```

6. Stop the embedded server.

   When you stop an embedded server, you also stop any containers and shards that are running on the server. When you stop an embedded server, you must clean up all open connections and move or tear down all the shards. The following example demonstrates how to stop a server and using the waitFor method on the Server interface to ensure that the Server instance shuts down completely. Similarly to the container example, the stopServer method is made available through the ServerMBean interface. With this interface, you can stop a server with the corresponding Managed Bean (MBean).

```
ServerFactory.stopServer();  // Uses the factory to kill the Server singleton
 // or
server.stopServer(); // Uses the Server instance directly
server.waitFor(); // Returns when the server has properly completed
         // its shutdown procedures
```

Full code example:

```
 import java.net.MalformedURLException;
import java.net.URL;

import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.deployment.DeploymentPolicy;
import com.ibm.websphere.objectgrid.deployment.DeploymentPolicyFactory;
import com.ibm.websphere.objectgrid.server.Container;
import com.ibm.websphere.objectgrid.server.Server;
import com.ibm.websphere.objectgrid.server.ServerFactory;
import com.ibm.websphere.objectgrid.server.ServerProperties;

public class ServerFactoryTest {

    public static void main(String[] args) {

        try {

            ServerProperties props = ServerFactory.getServerProperties();
            props.setCatalogServiceBootstrap("catalogservice-hostname:catalogservice-port");
            props.setServerName("ServerOne"); // name server
            props.setTraceSpecification("com.ibm.ws.objectgrid=all=enabled"); // TraceSpec

            /*
             * In most cases, the server will serve as a container server only
             * and will connect to an external catalog service. This is a more
             * highly available way of doing things. The commented code excerpt
             * below will enable this Server to be a catalog service.
             *
             *
             * CatalogServerProperties catalogProps =
             * ServerFactory.getCatalogProperties();
             * catalogProps.setCatalogServer(true); // enable catalog service
             * catalogProps.setQuorum(true); // enable quorum
             */

            Server server = ServerFactory.getInstance();

            DeploymentPolicy policy = DeploymentPolicyFactory.createDeploymentPolicy
     (new URL("url to deployment xml"),
                            new URL("url to objectgrid xml file"));
            Container container = server.createContainer(policy);

            /*
             * Shard will now be placed on this container if the deployment requirements are met.
             * This encompasses embedded server and container creation.
             *
             * The lines below will simply demonstrate calling the cleanup methods
             */

            container.teardown();
            server.stopServer();
            int success = server.waitFor();

        } catch (ObjectGridException e) {
            // Container failed to initialize
        } catch (MalformedURLException e2) {
            // invalid url to xml file(s)
        }

    }

}
```

Monitoring with the statistics API

The Statistics API is the direct interface to the internal statistics tree. Statistics are disabled by default, but can be enabled by setting a StatsSpec interface. A StatsSpec interface defines how WebSphere eXtreme Scale should monitor statistics.

About this task

You can use the local StatsAccessor API to query data and access statistics on any ObjectGrid instance that is in the same Java virtual machine (JVM) as the running code. For more information about the specific interfaces, see the API documentation. Use the following steps to enable monitoring of the internal statistics tree.

Procedure

1. Retrieve the StatsAccessor object. The StatsAccessor interface follows the singleton pattern. So, apart from problems related to the classloader, one StatsAccessor instance should exist for each JVM. This class serves as the main interface for all local statistics operations. The following code is an example of how to retrieve the accessor class. Call this operation before any other ObjectGrid calls.

```
public class LocalClient
{

   public static void main(String[] args) {

      // retrieve a handle to the StatsAccessor
      StatsAccessor accessor = StatsAccessorFactory.getStatsAccessor();

   }

}
```

2. Set the data grid StatsSpec interface. Set this JVM to collect all statistics at the ObjectGrid level only. You must ensure that an application enables all statistics that might be needed before you begin any transactions. The following example sets the StatsSpec interface using both a static constant field and using a spec String. Using a static constant field is simpler because the field has already defined the specification. However, by using a spec String, you can enable any combination of statistics that are required.

```
public static void main(String[] args) {

      // retrieve a handle to the StatsAccessor
      StatsAccessor accessor = StatsAccessorFactory.getStatsAccessor();

      // Set the spec via the static field
      StatsSpec spec = new StatsSpec(StatsSpec.OG_ALL);
      accessor.setStatsSpec(spec);

      // Set the spec via the spec String
      StatsSpec spec = new StatsSpec("og.all=enabled");
      accessor.setStatsSpec(spec);

}
```

3. Send transactions to the data grid to force data to be collected for monitoring. To collect useful data for statistics, you must send transactions to the data grid. The following code excerpt inserts a record into MapA, which is in ObjectGridA. Because the statistics are at the ObjectGrid level, any map within the ObjectGrid yields the same results.

```
public static void main(String[] args) {

      // retrieve a handle to the StatsAccessor
      StatsAccessor accessor = StatsAccessorFactory.getStatsAccessor();

      // Set the spec via the static field
      StatsSpec spec = new StatsSpec(StatsSpec.OG_ALL);
      accessor.setStatsSpec(spec);
      ObjectGridManager manager =
    ObjectGridmanagerFactory.getObjectGridManager();
      ObjectGrid grid = manager.getObjectGrid("ObjectGridA");
      Session session = grid.getSession();
      Map map = session.getMap("MapA");

      // Drive insert
      session.begin();
      map.insert("SomeKey", "SomeValue");
      session.commit();
}
```

4. Query a StatsFact by using the StatsAccessor API. Every statistics path is associated with a StatsFact interface. The StatsFact interface is a generic placeholder that is used to organize and contain a StatsModule object. Before you can access the actual statistics module, the StatsFact object must be retrieved.

```
public static void main(String[] args)
{

      // retrieve a handle to the StatsAccessor
      StatsAccessor accessor = StatsAccessorFactory.getStatsAccessor();

      // Set the spec via the static field
      StatsSpec spec = new StatsSpec(StatsSpec.OG_ALL);
      accessor.setStatsSpec(spec);

      ObjectGridManager manager =
    ObjectGridManagerFactory.getObjectGridManager();
      ObjectGrid grid = manager.getObjectGrid("ObjectGridA");
      Session session = grid.getSession();
      Map map = session.getMap("MapA");

      // Drive insert
      session.begin();
      map.insert("SomeKey", "SomeValue");
      session.commit();

      // Retrieve StatsFact

      StatsFact fact = accessor.getStatsFact(new String[] {"EmployeeGrid"},
    StatsModule.MODULE_TYPE_OBJECT_GRID);

}
```

5. Interact with the StatsModule object. The StatsModule object is contained within the StatsFact interface. You can obtain a reference to the module by using the StatsFact interface. Since the StatsFact interface is a generic interface, you must cast the returned module to the expected StatsModule type. Because this task collects eXtreme Scale statistics, the returned StatsModule object is cast to an OGStatsModule type. After the module is cast, you have access to all of the available statistics.

```
public static void main(String[] args) {

      // retrieve a handle to the StatsAccessor
      StatsAccessor accessor = StatsAccessorFactory.getStatsAccessor();

      // Set the spec via the static field
      StatsSpec spec = new StatsSpec(StatsSpec.OG_ALL);
      accessor.setStatsSpec(spec);

      ObjectGridManager manager =
    ObjectGridmanagerFactory.getObjectGridManager();
      ObjectGrid grid = manager.getObjectGrid("ObjectGridA");
      Session session = grid.getSession();
      Map map = session.getMap("MapA");

      // Drive insert
      session.begin();
      map.insert("SomeKey", "SomeValue");
      session.commit();

      // Retrieve StatsFact
      StatsFact fact = accessor.getStatsFact(new String[] {"EmployeeGrid"},
    StatsModule.MODULE_TYPE_OBJECT_GRID);

      // Retrieve module and time
      OGStatsModule module = (OGStatsModule)fact.getStatsModule();
      ActiveTimeStatistic timeStat =
    module.getTransactionTime("Default", true);
      double time = timeStat.getMeanTime();

}
```

Chapter 6. Integrating with Spring framework

Spring is a popular framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage eXtreme Scale transactions and configure the clients and servers comprising your deployed in-memory data grid.

Spring managed native transactions

Spring provides container-managed transactions that are similar to a Java Platform, Enterprise Edition application server. However, the Spring mechanism can plug in different implementations. WebSphere eXtreme Scale provides transaction manager integration which allows Spring to manage the ObjectGrid transaction life cycles. See the information about native transactions in the *Programming Guide* for details.

Spring managed extension beans and namespace support

Also, eXtreme Scale integrates with Spring to allow Spring-style beans defined for extension points or plug-ins. This feature provides more sophisticated configurations and more flexibility for configuring the extension points.

In addition to Spring managed extension beans, eXtreme Scale provides a Spring namespace called "objectgrid". Beans and built-in implementations are pre-defined in this namespace, which makes it easier for users to configure eXtreme Scale. Refer to “Spring extension beans and namespace support” on page 216 for more details on these topics and a sample of how to start an eXtreme Scale container server using Spring configurations.

Shard scope support

With the traditional style Spring configuration, an ObjectGrid bean can either be a singleton type or prototype type. ObjectGrid also supports a new scope called the "shard" scope. If a bean is defined as shard scope, then only one bean is created per shard. All requests for beans with an ID or ids matching that bean definition in the same shard will result in that one specific bean instance being returned by the Spring container.

The following example shows that a com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl bean is defined with scope set to shard. Therefore, only one instance of the JPAPropFactoryImpl class is created per shard.

```
<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard" />
```

Spring Web Flow

Spring Web Flow stores its session state in the HTTP Session by default. If a web application is configured to use eXtreme Scale for session management then it is used automatically by Spring to store this state and it is made fault tolerant in the same manner as the session.

Packaging

The eXtreme Scale Spring extensions are in the `ogspring.jar` file. This Java archive (JAR) file must be on the class path for Spring support to work. If a JEE application that is running in a WebSphere Extended Deployment augmented WebSphere Application Server Network Deployment, then the application should place the `spring.jar` file and its associated files in the enterprise archive (EAR) modules. You must also place the `ogspring.jar` file in the same location.

Native transactions

Spring is a popular framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage eXtreme Scale transactions and configure eXtreme Scale clients and servers.

Native transactions with WebSphere eXtreme Scale

Spring provides container-managed transactions along the style of a Java Platform, Enterprise Edition application server but has the advantage that Springs mechanism can have different implementations plugged in. This topic describes an eXtreme Scale Platform Transaction manager that can be used with Spring. This allows programmers to annotate their POJOs (plain old Java objects) and then have Spring automatically acquire Sessions from eXtreme Scale and begin, commit, rollback, suspend, and resume eXtreme Scale transactions. Spring transactions are described more fully in Chapter 10 of the official Spring User Guide. The following explains how to create an eXtreme Scale transaction manager and use it with annotated POJOs. It also explains how to use this approach with client or local eXtreme Scale as well as a collocated Data Grid style application.

Creating a transaction manager

eXtreme Scale provides an implementation of a Spring PlatformTransactionManager. This manager can provide managed eXtreme Scale sessions to POJOs managed by Spring. Through the use of annotations, Spring manages those sessions for the POJOs in terms of transaction life cycle. The following XML snippet shows how to create a transaction Manager:

```
<aop:aspectj-autoproxy/>
 <tx:annotation-driven transaction-manager="transactionManager"/>

 <bean id="ObjectGridManager"
    class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
    factory-method="getObjectGridManager"/>

 <bean id="ObjectGrid"
    factory-bean="ObjectGridManager"
    factory-method="createObjectGrid"/>

 <bean id="transactionManager"
    class="com.ibm.websphere.objectgrid.spring.ObjectGridSpringFactory"
    factory-method="getLocalPlatformTransactionManager"/>
 </bean>

 <bean id="Service" class="com.ibm.websphere.objectgrid.spring.test.TestService">
  <property name="txManager" ref+"transactionManager"/>
 </bean>
```

This shows the transactionManager bean being declared and wired in to the Service bean that will use Spring transactions. We will demonstrate this using annotations and this is the reason for the tx:annotation clause at the beginning.

Obtaining an ObjectGrid Session for the current Spring transaction

A POJO that has methods managed by Spring can now obtain the ObjectGrid session for the current transaction using

```
Session s = txManager.getSession();
```

This returns the session for the POJO to use. Beans participating in the same transaction will receive the same session when they call this method. Spring will automatically handle begin for the Session and also automatically invoke commit or rollback when necessary. You can obtain an ObjectGrid EntityManager also by simply calling getEntityManager from the Session object.

A sample POJO using annotations

Here is a POJO that uses annotations to declare its transactional intentions to Spring. You can see the class has a class level annotation indicating all methods by default should use REQUIRED transaction semantics. The class implements an interface with a method for all methods on the class. This is necessary for Spring AOP to work when it cannot do bytecode weaving. The class has an instance variable txManager that we wire to the ObjectGrid transaction manager using the Spring xml file. Each method simply calls the txManager.getSession method to obtain the Session to use for the method.The queryNewTx method is annotated to indicate a REQUIRES_NEW semantic. This means any existing transaction will be suspended and a new independent transaction created for that method.

```
@Transactional(propagation=Propagation.REQUIRED)
public class TestService implements ITestService
{
    SpringLocalTxManager txManager;

    public TestService()
    {

    }

    public void initialize()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        ObjectMap m = s.getMap("TEST");
        m.insert("Hello", "Billy");
    }

    public void update(String updatedValue)
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        System.out.println("Update using " + s);
        ObjectMap m = s.getMap("TEST");
        String v = (String)m.get("Hello");
        m.update("Hello", updatedValue);
    }

    public String query()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        System.out.println("Query using " + s);
        ObjectMap m = s.getMap("TEST");
        return (String)m.get("Hello");
    }

    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public String queryNewTx()
        throws ObjectGridException
    {
        Session s = txManager.getSession();
        System.out.println("QueryTX using " + s);
        ObjectMap m = s.getMap("TEST");
        return (String)m.get("Hello");
    }

    public void testRequiresNew(ITestService bean)
        throws ObjectGridException
    {
        update("1");
        String txValue = bean.query();
        if(!txValue.equals("1"))
        {
            System.out.println("Requires didnt work");
            throw new IllegalStateException("requires didn't work");
        }
        String committedValue = bean.queryNewTx();
        if(committedValue.equals("1"))
        {
            System.out.println("Requires new didnt work");
            throw new IllegalStateException("requires new didn't work");
        }
    }

    public SpringLocalTxManager getTxManager() {
        return txManager;
    }

    public void setTxManager(SpringLocalTxManager txManager) {
        this.txManager = txManager;
    }
}
```

Setting the ObjectGrid instance for a thread

A single Java Virtual Machine (JVM) can host many ObjectGrid instances. Each primary shard placed in a JVM has its own ObjectGrid instance. A JVM acting as a client to a remote ObjectGrid uses an ObjectGrid instance returned from the connect method's ClientClusterContext to interact with that Grid. Before invoking a method on a POJO using Spring transactions for ObjectGrid, the thread must be primed with the ObjectGrid instance to use. The TransactionManager instance has a method allowing a specific ObjectGrid instance to be specified. Once specified then any subsequent txManager.getSession calls will returns Sessions for that ObjectGrid instance.

Simple bootstrap for testing

The following example shows a sample main for exercising this capability:

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(new String[]
            {"applicationContext.xml"});
    SpringLocalTxManager txManager = (SpringLocalTxManager)ctx.getBean("transactionManager");
    txManager.setObjectGridForThread(og);

    ITestService s = (ITestService)ctx.getBean("Service");
    s.initialize();
    assertEquals(s.query(), "Billy");
    s.update("Bobby");
    assertEquals(s.query(), "Bobby");
    System.out.println("Requires new test");
    s.testRequiresNew(s);
    assertEquals(s.query(), "1");
```

Here we use a Spring ApplicationContext. The ApplicationContext is used to obtain a reference to the txManager and specify an ObjectGrid to use on this thread. The code then obtains a reference to the service and invokes methods on it. Each method call at this level causes Spring to create a Session and do begin/commit calls around the method call. Any exceptions will cause a rollback.

New eXtreme Scale interfaces

This introduces a single new interface, SpringLocalTxManager. This interface is implemented by the ObjectGrid Platform Transaction Manager and has all public interfaces. The methods on this interface are for selecting the ObjectGrid instance to use on a thread and obtaining a Session for the thread. Any POJOs using ObjectGrid local transactions should be injected with a reference to this manager instance and only a single instance need be created, that is, its scope should be singleton.This instance is created using a static method on ObjectGridSpringFactory. getLocalPlatformTransactionManager().

eXtreme Scale for JTA and global transactions

eXtreme Scale does not support JTA or 2 phase commit for various reasons mainly to do with scalability. Thus, except at a last single-phase participant, ObjectGrid does not interact in XA or JTA type global transactions. This platform manager is intended to make using local ObjectGrid transactions as easy as possible for Spring developers.

Spring managed extension beans

ObjectGrid includes an approach to declare POJOs to use as extension points in the objectgrid.xml file. ObjectGrid provides a way to name the beans and then specify the class name. ObjectGrid normally creates instances of the specified class and uses those instances as the plug-in. ObjectGrid can now delegate to Spring to act as the bean factory for obtaining instances of these plug-in objects. If an application uses Spring then typically such POJOs have a requirement to be wired in to the rest of the application.

ObjectGrid has been modified to allow an application to register a Spring Bean Factory instance to use for a specific named ObjectGrid. The application should create an instance of BeanFactory or a Spring application context and then register it with ObjectGrid using the following static method:

```
void registerSpringBeanFactoryAdapter(String objectGridName, Object springBeanFactory)
```

This method specifies that if ObjectGrid finds an extension bean (such as an ObjectTransformer, Loader, TransactionCallback, and so on) whose className begins with the prefix {spring} then use the remainder of the name as a Spring Bean name and obtain the bean instance using the Spring Bean Factory. ObjectGrid can also create a Spring bean factory from a default spring xml configuration file. If no bean factory was registered for a given ObjectGrid then ObjectGrid tries to find an xml file called 'ObjectGridName'_spring.xml. For example, if your grid is called GRID then the xml file is called '/GRID_spring.xml' and should be in the class path in the root package. If this file is found then ObjectGrid constructs an ApplicationContext using that file and constructs beans from that bean factory. As example class name would be:

```
"{spring}MyLoaderBean"
```

This would cause ObjectGrid to ask Spring for a bean named "MyLoaderBean". This approach can be used to specify Spring managed POJOs for any extension point in ObjectGrid so long as the bean factory has been registered up front. The ObjectGrid spring extensions are in the ogspring.jar file. This jar file must be on the class path for spring support to work due. If a JavaEE application running in an XD augmented ND then the application should place the spring.jar file and its associated files in the EAR modules. The ogspring.jar must also be placed in the same location.

Spring Webflow

Spring Webflow stores its session state in the HTTP Session by default. If a web application is configured to use ObjectGrid for session management then ObjectGrid will be used automatically by Spring to store this state and it will be made fault tolerant in the same manner as the session.

Spring extension beans and namespace support

WebSphere eXtreme Scale provides a feature to declare plain old Java objects (POJOs) to use as extension points in the `objectgrid.xml` file and a way to name the beans and then specify the class name. Normally, instances of the specified class are created, and those objects are used as the plug-ins. Now, eXtreme Scale can delegate to Spring to obtain instances of these plug-in objects. If an application uses Spring then typically such POJOs have a requirement to be wired in to the rest of the application.

In some cases, you must use Spring to configure certain plug-in objects. Take the following configuration as an example:

```
<objectGrid name="Grid">
    <bean id="TransactionCallback" className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
        <property name="persistenceUnitName" type="java.lang.String"  value="employeePU" />
    </bean>
    ...
</objectGrid>
```

The built-in TransactionCallback implementation, com.ibm.websphere.objectgrid.jpa.JPATxCallback class, is configured as the TransactionCallback class. This class is configured with one property persistenceUnitName as shown in the previous example. The JPATxCallback class also has the JPAPropertyFactory attribute, which is of type java.lang.Object. The ObjectGrid XML configuration cannot support this type of configuration.

The eXtreme Scale Spring integration solves this problem by delegating the bean creation to the Spring framework. The revised configuration follows:

```
<objectGrid name="Grid">
    <bean id="TransactionCallback" className="{spring}jpaTxCallback"/>
    ...
</objectGrid>
```

The spring file for the "Grid" object contains the following information:

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
    <property name="persistenceUnitName" value="employeeEMPU"/>
    <property name="JPAPropertyFactory" ref ="jpaPropFactory"/>
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.
JPAPropFactoryImpl" scope="shard">
</bean>
```

Here, the TransactionCallback is specified as `{spring}jpaTxCallback`, and the jpaTxCallback and jpaPropFactory bean are configured in the spring file as shown in the previous example. The Spring configuration makes it possible to configure a JPAPropertyFactory bean as a parameter of the JPATxCallback object.

Default Spring bean factory

When eXtreme Scale finds a plug-in or an extension bean (such as an ObjectTransformer, Loader, TransactionCallback, and so on) with a classname value that begins with the prefix `{spring}`, then eXtreme Scale uses the remainder of the name as a Spring Bean name and obtain the bean instance using the Spring Bean Factory.

By default, if no bean factory was registered for a given ObjectGrid, then it tries to find an `ObjectGridName_spring.xml` file. For example, if your data grid is called "Grid" then the XML file is called `/Grid_spring.xml`. This file should be in the class path or in a `META-INF` directory which is in the class path. If this file is found, then eXtreme Scale constructs an ApplicationContext using that file and constructs beans from that bean factory.

Custom Spring bean factory

WebSphere eXtreme Scale also provides an ObjectGridSpringFactory API to register a Spring Bean Factory instance to use for a specific named ObjectGrid. This API registers an instance of BeanFactory with eXtreme Scale using the following static method:

```
void registerSpringBeanFactoryAdapter(String objectGridName, Object
springBeanFactory)
```

Namespace support

Since version 2.0, Spring has a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. ObjectGrid uses this new feature to define and configure ObjectGrid beans. With Spring XML schema extension, some of the built-in implementations of eXtreme Scale plug-ins and some ObjectGrid beans are predefined in the "objectgrid" namespace. When writing the Spring configuration files, you do not have to specify the full class name of the built-ins. Instead, you can reference the predefined beans.

Also, with the attributes of the beans defined in the XML schema, you are less likely to provide a wrong attribute name. XML validation based on the XML schema can catch these kind of errors earlier in the development cycle.

These beans defined in the XML schema extensions are:
- transactionManager
- register
- server
- catalog
- container
- JPALoader
- JPATxCallback
- JPAEntityLoader
- LRUEvictor
- LFUEvictor
- HashIndex

These beans are defined in the objectgrid.xsd XML schema. This XSD file is shipped as `com/ibm/ws/objectgrid/spring/namespace/objectgrid.xsd` file in the `ogspring.jar` file . For detailed descriptions of the XSD file and the beans defined in the XSD file, see the information about the Spring descriptor file in the *Administration Guide*.

Still use the JPATxCallback example from the previous section. In the previous section, the JPATxCallback bean is configured as the following:

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
    <property name="persistenceUnitName" value="employeeEMPU"/>
    <property name="JPAPropertyFactory" ref ="jpaPropFactory"/>
```

```
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard">
</bean>
```

Using this namespace feature, the spring XML configuration can be written as the following:

```
<objectgrid:JPATxCallback id="jpaTxCallback" persistenceUnitName="employeeEMPU"
 jpaPropertyFactory="jpaPropFactory" />

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl"
 scope="shard">
</bean>
```

Notice here that instead of specifying the "com.ibm.websphere.objectgrid.jpa.JPATxCallback" class as in the previous example, we directly use the pre-defined "objectgrid:JPATxCallback" bean. As you can see, this configuration is less verbose and more friendly to error checking.

Starting container server with Spring extension beans

In this example, we will show how to start an ObjectGrid server using ObjectGrid Spring managed extension beans and namespace support.

ObjectGrid XML file

First of all, define a very simple ObjectGrid XML file which contains one ObjectGrid "Grid" and one map "Test". The ObjectGrid has an ObjectGridEventListener plug-in called "partitionListener", and the map "Test" has an Evictor plugged in called "testLRUEvictor". Notice both the ObjectGridEventListener plug-in and Evictor plug-in are configured using Spring as their names contain "{spring}".

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="Grid">
     <bean id="ObjectGridEventListener" className="{spring}partitionListener" />
            <backingMap name="Test" pluginCollectionRef="test" />
        </objectGrid>
    </objectGrids>

    <backingMapPluginCollections>
        <backingMapPluginCollection id="test">
            <bean id="Evictor" className="{spring}testLRUEvictor"/>
        </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

ObjectGrid deployment XML file

Now, create a simple ObjectGrid deployment XML file as follows. It partitions the ObjectGrid into 5 partitions, and no replica is required.

```
<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy ../deploymentPolicy.xsd"
 xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">
    <objectgridDeployment objectgridName="Grid">
        <mapSet name="mapSet" numInitialContainers="1" numberOfPartitions="5" minSyncReplicas="0"
            maxSyncReplicas="1" maxAsyncReplicas="0">
            <map ref="Test"/>
        </mapSet>
    </objectgridDeployment>
</deploymentPolicy>
```

ObjectGrid Spring XML file

Now we will use both ObjectGrid Spring managed extension beans and namespace support features to configure the ObjectGrid beans. The spring xml file is named "Grid_spring.xml". Notice two schemas are included in the XML file: spring-beans-2.0.xsd is for using the Spring managed beans, and objectgrid.xsd is for using the beans predefined in the objectgrid namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"
       xsi:schemaLocation="
       http://www.ibm.com/schema/objectgrid
   http://www.ibm.com/schema/objectgrid/objectgrid.xsd
       http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <objectgrid:register id="ogregister" gridname="Grid"/>

    <objectgrid:server id="server" isCatalog="true" name="server">
        <objectgrid:catalog host="localhost" port="2809"/>
    </objectgrid:server>

    <objectgrid:container id="container"
   objectgridxml="com/ibm/ws/objectgrid/test/springshard/objectgrid.xml"
       deploymentxml="com/ibm/ws/objectgrid/test/springshard/deployment.xml"
   server="server"/>

    <objectgrid:LRUEvictor id="testLRUEvictor" numberOfLRUQueues="31"/>

    <bean id="partitionListener"
   class="com.ibm.websphere.objectgrid.springshard.ShardListener" scope="shard"/>
</beans>
```

There were 6 beans defined in this spring XML file:

1. *objectgrid:register*: This register the default bean factory for the ObjectGrid "Grid".
2. *objectgrid:server*: This defines an ObjectGrid server with name "server". This server will also provide catalog service since it has an objectgrid:catalog bean nested in it.
3. *objectgrid:catalog*: This defines an ObjectGrid catalog service endpoint, which is set to "localhost:2809".
4. *objectgrid:container*: This defines an ObjectGrid container with specified objectgrid XML file and deployment XML file as we discussed before. The server property specifies which server this container is hosted in.
5. *objectgrid:LRUEvictor*: This defines an LRUEvictor with the number of LRU queues to use set to 31.
6. *bean partitionListener*: This defines a ShardListener plug-in. This class is a class plugged in by users, so it cannot use the pre-defined beans. Also this scope of the bean is set to "shard", which means there is only one instance of this ShardListener per ObjectGrid shard.

Starting the server

The snippet below starts the ObjectGrid server, which hosts both the container service and the catalog service. As we can see, the only method we need to call to start the server is to get a bean "container" from the bean factory. This simplifies the programming complexity by moving most of the logic into Spring configuration.

```
public class ShardServer extends TestCase
{
    Container container;
    org.springframework.beans.factory.BeanFactory bf;

    public void startServer(String cep)
    {
        try
        {
            bf = new org.springframework.context.support.ClassPathXmlApplicationContext(
                "/com/ibm/ws/objectgrid/test/springshard/Grid_spring.xml", ShardServer.class);
            container = (Container)bf.getBean("container");
        }
        catch(Exception e)
        {
            throw new ObjectGridRuntimeException("Cannot start OG container", e);
        }
    }

    public void stopServer()
    {
        if(container != null)
            container.teardown();
    }
}
```

Chapter 7. Security API

WebSphere eXtreme Scale adopts an open security architecture. It provides a basic security framework for authentication, authorization, and transport security, and requires users to implement plug-ins to complete the security infrastructure.

The following image shows the basic flow of client authentication and authorization for an eXtreme Scale server.

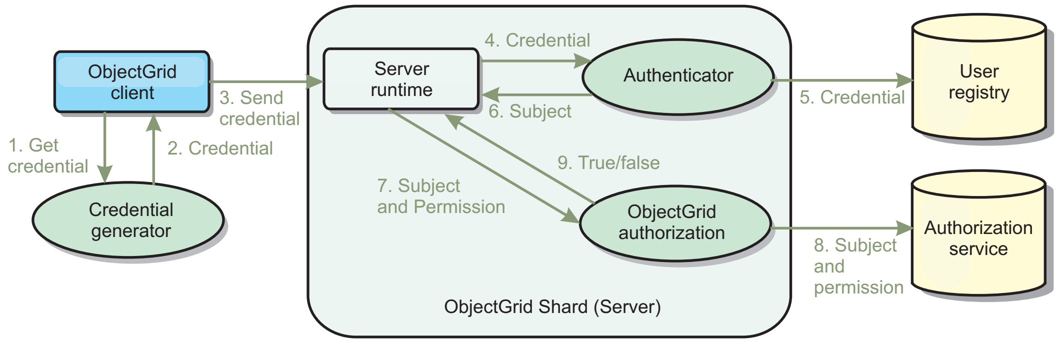

Figure 6. Flow of client authentication and authorization

The authentication flow and authorization flow are as follows.

Authentication flow

1. The authentication flow starts with an eXtreme Scale client getting a credential. This is done by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator plug-in.
2. A CredentialGenerator object knows how to generate a valid client credential, for example, a user ID and password pair, Kerberos ticket, and so on. This generated credential is sent back to the client.
3. After the client retrieves the Credential object using the CredentialGenerator object, this Credential object is sent along with the eXtreme Scale request to the eXtreme Scale server.
4. The eXtreme Scale server authenticates the Credential object before processing the eXtreme Scale request. Then the server uses the Authenticator plug-in to authenticate the Credential object.
5. The Authenticator plug-in represents an interface to the user registry, for example, a Lightweight Directory Access Protocol (LDAP) server or an operating system user registry. The Authenticator consults the user registry and makes authentication decisions.
6. If the authentication is successful, a Subject object is returned to represent this client.

   Authorization flow

   WebSphere eXtreme Scale adopts a permission-based authorization mechanism, and has different permission categories represented by different permission classes. For example, a com.ibm.websphere.objectgrid.security.MapPermission object represents permissions to read, write, insert, invalidate, and remove the data entries in an ObjectMap. Because WebSphere eXtreme Scale supports Java

Authentication and Authorization Service (JAAS) authorization out-of-box, you can use JAAS to handle authorization by providing authorization policies.

Also, eXtreme Scale supports custom authorizations. Custom authorizations are plugged in by the plug-in com.ibm.websphere.objectgrid.security.plugins.ObjectGridAuthorization. The flow of the customer authorization is as follows.

7. The server runtime sends the Subject object and the required permission to the authorization plug-in.
8. The authorization plug-in consults the Authorization service and makes an authorization decision. If permission is granted for this Subject object, a value of `true` is returned, otherwise `false`is returned.
9. This authorization decision, true or false, is returned to the server runtime.

Security implementation

The topics in this section discuss how to program a secure WebSphere eXtreme Scale deployment and how to program the plug-in implementations. The section is organized based on the various security features. In each subtopic, you will learn about relevant plug-ins and how to implement the plug-ins. In the authentication section, you will see how to connect to a secure WebSphere eXtreme Scale deployment environment.

Client Authentication: The client authentication topic describes how a WebSphere eXtreme Scale client gets a credential and how a server authenticates the client. It will also discuss how a WebSphere eXtreme Scale client connects to a secure WebSphere eXtreme Scale server.

Authorization: The authorization topic explains how to use the ObjectGridAuthorization to do customer authorization besides JAAS authorization.

Grid Authentication: The data grid authentication topic discusses how you can use SecureTokenManager to securely transport server secrets.

Java Management Extensions (JMX) programming: When the WebSphere eXtreme Scale server is secured, the JMX client might need to send a JMX credential to the server.

Client authentication programming

For authentication, WebSphere eXtreme Scale provides a runtime to send the credential from the client to the server side, and then calls the authenticator plug-in to authenticate the users.

WebSphere eXtreme Scale requires you to implement the following plug-ins to complete the authentication.

- Credential: A Credential represents a client credential, such as a user ID and password pair.
- CredentialGenerator: A CredentialGenerator represents a credential factory to generate the credential.
- Authenticator: An Authenticator authenticates the client credential and retrieves client information.

Credential and CredentialGenerator plug-ins

When an eXtreme Scale client connects to a server that requires authentication, the client is required to provide a client credential. A client credential is represented by a com.ibm.websphere.objectgrid.security.plugins.Credential interface. A client credential can be a user name and password pair, a Kerberos ticket, a client certificate, or data in any format that the client and server agree upon. See the information about the Credential API in the API documentation for more details. This interface explicitly defines the equals(Object) and hashCode methods. These two methods are important because the authenticated Subject objects are cached by using the Credential object as the key on the server side. WebSphere eXtreme Scale also provides a plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface and is useful when the credential is can expire. In this case, the getCredential method is called to renew a credential.

The Credential interface explicitly defines the equals(Object) and hashCode methods. These two methods are important because the authenticated Subject objects are cached by using the Credential object as the key on the server side.

You may also use the provided plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface, and is useful when the credential can expire. In this case, the getCredential method is called to renew a credential. See the API documentation for more details.

There are three provided default implementations for the Credential interfaces:
- The com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential implementation, which contains a user ID and password pair.
- The com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential implementation, which contains WebSphere Application Server-specific authentication and authorization tokens. These tokens can be used to propagate the security attributes across the application servers in the same security domain.

WebSphere eXtreme Scale also provides a plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface.WebSphere eXtreme Scale provides two default built-in implementations:
- The com.ibm.websphere.objectgrid.security.plugins.builtins. UserPasswordCredentialGenerator constructor takes a user ID and a password. When the getCredential method is called, it returns a UserPasswordCredential object that contains the user ID and password.
- The com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator represents a credential (security token) generator when running in WebSphere Application Server. When the getCredential method is called, the Subject that is associated with the current thread is retrieved. Then the security information in this Subject object is converted into a WSTokenCredential object. You can specify whether to retrieve a runAs subject or a caller subject from the thread by using the constant WSTokenCredentialGenerator.RUN_AS_SUBJECT or WSTokenCredentialGenerator.CALLER_SUBJECT.

UserPasswordCredential and UserPasswordCredentialGenerator

For testing purposes, WebSphere eXtreme Scale provides the following plug-in implementations:

1. `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential`
2. `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator`

The user password credential stores a user ID and password. The user password credential generator then contains this user ID and password.

The following example code shows how to implement these two plug-ins.

UserPasswordCredential.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import com.ibm.websphere.objectgrid.security.plugins.Credential;

/**
 * This class represents a credential containing a user ID and password.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Credential
 * @see UserPasswordCredentialGenerator#getCredential()
 */
public class UserPasswordCredential implements Credential {

    private static final long serialVersionUID = 1409044825541007228L;

    private String ivUserName;

    private String ivPassword;

    /**
     * Creates a UserPasswordCredential with the specified user name and
     * password.
     *
     * @param userName the user name for this credential
     * @param password the password for this credential
     *
     * @throws IllegalArgumentException if userName or password is <code>null</code>
     */
    public UserPasswordCredential(String userName, String password) {
        super();
        if (userName == null || password == null) {
            throw new IllegalArgumentException("User name and password cannot be null.");
        }
        this.ivUserName = userName;
        this.ivPassword = password;
    }

    /**
     * Gets the user name for this credential.
     *
     * @return   the user name argument that was passed to the constructor
     *           or the <code>setUserName(String)</code>
     *           method of this class
     *
     * @see #setUserName(String)
     */
    public String getUserName() {
        return ivUserName;
    }

    /**
     * Sets the user name for this credential.
     *
     * @param userName the user name to set.
     *
     * @throws IllegalArgumentException if userName is <code>null</code>
     */
    public void setUserName(String userName) {
        if (userName == null) {
            throw new IllegalArgumentException("User name cannot be null.");
        }
        this.ivUserName = userName;
    }
    /**
     * Gets the password for this credential.
     *
     * @return   the password argument that was passed to the constructor
     *           or the <code>setPassword(String)</code>
     *           method of this class
     *
     * @see #setPassword(String)
     */
    public String getPassword() {
        return ivPassword;
    }

    /**
     * Sets the password for this credential.
     *
     * @param password the password to set.
     *
     * @throws IllegalArgumentException if password is <code>null</code>
     */
    public void setPassword(String password) {
        if (password == null) {
            throw new IllegalArgumentException("Password cannot be null.");
        }
        this.ivPassword = password;
    }

    /**
     * Checks two UserPasswordCredential objects for equality.
     * <p>
     * Two UserPasswordCredential objects are equal if and only if their user names
     * and passwords are equal.
     *
     * @param o the object we are testing for equality with this object.
     *
     * @return <code>true</code> if both UserPasswordCredential objects are equivalent.
     *
     * @see Credential#equals(Object)
     */
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o instanceof UserPasswordCredential) {
            UserPasswordCredential other = (UserPasswordCredential) o;
            return other.ivPassword.equals(ivPassword) && other.ivUserName.equals(ivUserName);
        }

        return false;
    }

    /**
     * Returns the hashcode of the UserPasswordCredential object.
     *
     * @return the hash code of this object
     *
     * @see Credential#hashCode()
     */
    public int hashCode() {
        return ivUserName.hashCode() + ivPassword.hashCode();
    }
}
```

UserPasswordCredentialGenerator.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.util.StringTokenizer;

import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator;

/**
 * This credential generator creates <code>UserPasswordCredential</code> objects.
 * <p>
 * UserPasswordCredentialGenerator has a one to one relationship with
 * UserPasswordCredential because it can only create a UserPasswordCredential
 * representing one identity.
 *
 * @since WAS XD 6.0.1
 * @ibm-api
 *
 * @see CredentialGenerator
 * @see UserPasswordCredential
 */
public class UserPasswordCredentialGenerator implements CredentialGenerator {

    private String ivUser;

    private String ivPwd;

    /**
     * Creates a UserPasswordCredentialGenerator with no user name or password.
     *
     * @see #setProperties(String)
     */
    public UserPasswordCredentialGenerator() {
        super();
    }

    /**
     * Creates a UserPasswordCredentialGenerator with a specified user name and
     * password
     *
     * @param user the user name
     * @param pwd the password
     */
    public UserPasswordCredentialGenerator(String user, String pwd) {
        ivUser = user;
        ivPwd = pwd;
    }

    /**
     * Creates a new <code>UserPasswordCredential</code> object using this
     * object's user name and password.
     *
     * @return a new <code>UserPasswordCredential</code> instance
     *
     * @see CredentialGenerator#getCredential()
     * @see UserPasswordCredential
     */
    public Credential getCredential() {
        return new UserPasswordCredential(ivUser, ivPwd);
    }

    /**
     * Gets the password for this credential generator.
     *
     * @return   the password argument that was passed to the constructor
     */
    public String getPassword() {
        return ivPwd;
    }

    /**
     * Gets the user name for this credential.
     *
     * @return   the user argument that was passed to the constructor
     *           of this class
     */
    public String getUserName() {
        return ivUser;
    }
    /**
     * Sets additional properties namely a user name and password.
     *
     * @param properties a properties string  with a user name and
     *                   a password separated by a blank.
     *
     * @throws IllegalArgumentException if the format is not valid
     */
    public void setProperties(String properties) {
        StringTokenizer token = new StringTokenizer(properties, " ");
        if (token.countTokens() != 2) {
            throw new IllegalArgumentException(
                "The properties should have a user name and password and separated by a blank.");
        }

        ivUser = token.nextToken();
        ivPwd = token.nextToken();
    }
    /**
     * Checks two UserPasswordCredentialGenerator objects for equality.
     * <p>
     * Two UserPasswordCredentialGenerator objects are equal if and only if
     * their user names and passwords are equal.
     *
     * @param obj the object we are testing for equality with this object.
     *
     * @return <code>true</code> if both UserPasswordCredentialGenerator objects
     *         are equivalent.
     */
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }

        if (obj != null && obj instanceof UserPasswordCredentialGenerator) {
            UserPasswordCredentialGenerator other = (UserPasswordCredentialGenerator) obj;

            boolean bothUserNull = false;
            boolean bothPwdNull = false;

            if (ivUser == null) {
                if (other.ivUser == null) {
                    bothUserNull = true;
                } else {
                    return false;
                }
            }

            if (ivPwd == null) {
                if (other.ivPwd == null) {
                    bothPwdNull = true;
                } else {
                    return false;
                }
            }

            return (bothUserNull || ivUser.equals(other.ivUser)) && (bothPwdNull || ivPwd.equals(other.ivPwd));
        }

        return false;
    }

    /**
     * Returns the hashcode of the UserPasswordCredentialGenerator object.
     *
     * @return the hash code of this object
     */
    public int hashCode() {

        return ivUser.hashCode() + ivPwd.hashCode();
    }

}
```

The UserPasswordCredential class contains two attributes: user name and password. The UserPasswordCredentialGenerator serves as a factory that contains the UserPasswordCredential objects.

WSTokenCredential and WSTokenCredentialGenerator

When the WebSphere eXtreme Scale clients and servers are all deployed in WebSphere Application Server, the client application can use these two built-in implementations when the following conditions are satisfied:

1. WebSphere Application Server global security is turned on.
2. All WebSphere eXtreme Scale clients and servers are running in WebSphere Application Server Java virtual machines.
3. The application servers are in the same security domain.
4. The client is already authenticated in WebSphere Application Server.

In this situation, the client can use the com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator class to generate a credential. The server uses the WSAuthenticator implementation class to authenticate the credential.

This scenario takes advantage of the fact that the eXtreme Scale client has already been authenticated. Because the application servers that have the servers are in the same security domain as the application servers that house the clients, the security tokens can be propagated from the client to the server so that the same user registry does not need to be authenticated again.

Note: Do not assume that a CredentialGenerator always generates the same credential. For an expirable and refreshable credential, the CredentialGenerator should be able to generate the latest valid credential to make sure the authentication succeeds. One example is using the Kerberos ticket as a Credential object. When the Kerberos ticket refreshes, the CredentialGenerator should retrieve the refreshed ticket when CredentialGenerator.getCredential is called.

Authenticator plug-in

After the eXtreme Scale client retrieves the Credential object using the CredentialGenerator object, this client Credential object is sent along with the client request to the eXtreme Scale server. The server authenticates the Credential object before processing the request. If the Credential object is authenticated successfully, a Subject object is returned to represent this client.

This Subject object is then cached, and it expires after its lifetime reaches the session timeout value. The login session timeout value can be set by using the loginSessionExpirationTime property in the cluster XML file. For example, setting `loginSessionExpirationTime="300"` makes the Subject object expire in 300 seconds.

This Subject object is then used for authorizing the request, which is shown later. An eXtreme Scale server uses the Authenticator plug-in to authenticate the Credential object. See the information about the Authenticator in the API documentation for more details.

The Authenticator plug-in is where the eXtreme Scale runtime authenticates the Credential object from the client user registry, for example, a Lightweight Directory Access Protocol (LDAP) server.

WebSphere eXtreme Scale does not provide an immediately available user registry configuration. The configuration and management of user registry is left outside of WebSphere eXtreme Scale for simplicity and flexibility. This plug-in implements connecting and authenticating to the user registry. For example, an Authenticator implementation extracts the user ID and password from the credential, uses them to connect and validate to an LDAP server, and creates a Subject object as a result of the authentication. The implementation might use JAAS login modules. A Subject object is returned as a result of authentication.

Notice that this method creates two exceptions: InvalidCredentialException and ExpiredCredentialException. The InvalidCredentialException exception indicates that the credential is not valid. The ExpiredCredentialException exception indicates that the credential expired. If one of these two exceptions result from the authenticate method, the exceptions are sent back to the client. However, the client runtime handles these two exceptions differently:

- If the error is an InvalidCredentialException exception, the client run time displays this exception. Your application must handle the exception. You can correct the CredentialGenerator, for example, and then retry the operation.
- If the error is an ExpiredCredentialException exception, and the retry count is not 0, the client run time calls the CredentialGenerator.getCredential method again, and sends the new Credential object to the server. If the new credential authentication succeeds, the server processes the request. If the new credential authentication fails, the exception is sent back to the client. If the number of authentication retries reaches the supported value and the client still gets an ExpiredCredentialException exception, the ExpiredCredentialException exception results. Your application must handle the error.

The Authenticator interface provides great flexibility. You can implement the Authenticator interface in your own specific way. For example, you can implement this interface to support two different user registries.

WebSphere eXtreme Scale provides sample authenticator plug-in implementations. Except for the WebSphere Application Server authenticator plug-in, the other implementations are only samples for testing purposes.

KeyStoreLoginAuthenticator

This example uses an eXtreme Scale built-in implementation: KeyStoreLoginAuthenticator, which is for testing and sample purposes (a key store is a simple user registry and should not be used for a production environment). Again, the class is displayed to further demonstrate how to implement an authenticator.

KeyStoreLoginAuthenticator.java
```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007

package com.ibm.websphere.objectgrid.security.plugins.builtins;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import com.ibm.websphere.objectgrid.security.plugins.Authenticator;
import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.ExpiredCredentialException;
import com.ibm.websphere.objectgrid.security.plugins.InvalidCredentialException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.security.auth.callback.UserPasswordCallbackHandlerImpl;

/**
 * This class is an implementation of the <code>Authenticator</code> interface
 * when a user name and password are used as a credential.
 * <p>
 * When user ID and password authentication is used, the credential passed to the
 * <code>authenticate(Credential)</code> method is a UserPasswordCredential object.
 * <p>
 * This implementation will use a <code>KeyStoreLoginModule</code> to authenticate
 * the user into the key store using the JAAS login module "KeyStoreLogin". The key
 * store can be configured as an option to the <code>KeyStoreLoginModule</code>
 * class. Please see the <code>KeyStoreLoginModule</code> class for more details
 * about how to set up the JAAS login configuration file.
 * <p>
 * This class is only for sample and quick testing purpose. Users should
 * write your own Authenticator implementation which can fit better into
 * the environment.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Authenticator
 * @see KeyStoreLoginModule
 * @see UserPasswordCredential
 */
public class KeyStoreLoginAuthenticator implements Authenticator {

    /**
     * Creates a new KeyStoreLoginAuthenticator.
     */
    public KeyStoreLoginAuthenticator() {
        super();
    }

    /**
     * Authenticates a <code>UserPasswordCredential</code>.
     * <p>
     * Uses the user name and password from the specified UserPasswordCredential
     * to login to the KeyStoreLoginModule named "KeyStoreLogin".
     *
     * @throws InvalidCredentialException if credential isn't a
     *         UserPasswordCredential or some error occurs during processing
     *         of the supplied UserPasswordCredential
     *
     * @throws ExpiredCredentialException if credential is expired.  This exception
     *         is not used by this implementation
     *
     * @see Authenticator#authenticate(Credential)
     * @see KeyStoreLoginModule
     */
    public Subject authenticate(Credential credential) throws InvalidCredentialException,
   ExpiredCredentialException {

        if (credential == null) {
            throw new InvalidCredentialException("Supplied credential is null");
        }

        if (! (credential instanceof UserPasswordCredential) ) {
            throw new InvalidCredentialException("Supplied credential is not a UserPasswordCredential");
        }

        UserPasswordCredential cred = (UserPasswordCredential) credential;
        LoginContext lc = null;
        try {
            lc = new LoginContext("KeyStoreLogin",
                    new UserPasswordCallbackHandlerImpl(cred.getUserName(), cred.getPassword().toCharArray()));

            lc.login();

            Subject subject = lc.getSubject();

            return subject;
        }
        catch (LoginException le) {
            throw new InvalidCredentialException(le);
        }
        catch (IllegalArgumentException ile) {
            throw new InvalidCredentialException(ile);
        }
    }
}
```

KeyStoreLoginModule.java

```
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.io.File;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import javax.security.auth.x500.X500Principal;
import javax.security.auth.x500.X500PrivateCredential;

import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.util.ObjectGridUtil;

/**
 * A KeyStoreLoginModule is keystore authentication login module based on
 * JAAS authentication.
 * <p>
 * A login configuration should provide an option "<code>keyStoreFile</code>" to
 * indicate where the keystore file is located. If the <code>keyStoreFile</code>
 * value contains a system property in the form, <code>${system.property}</code>,
 * it will be expanded to the value of the system property.
 * <p>
 * If an option "<code>keyStoreFile</code>" is not provided, the default keystore
 * file name is <code>"${java.home}${/}.keystore"</code>.
 * <p>
 * Here is a Login module configuration example:
 * <pre><code>
 *    KeyStoreLogin {
 *        com.ibm.websphere.objectgrid.security.plugins.builtins.KeystoreLoginModule required
 *            keyStoreFile="${user.dir}${/}security${/}.keystore";
 *    };
 * </code></pre>
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see LoginModule
 */
public class KeyStoreLoginModule implements LoginModule {

    private static final String CLASS_NAME = KeyStoreLoginModule.class.getName();

    /**
     * Key store file property name
     */
    public static final String KEY_STORE_FILE_PROPERTY_NAME = "keyStoreFile";

    /**
     * Key store type. Only JKS is supported
     */
    public static final String KEYSTORE_TYPE = "JKS";

    /**
     * The default key store file name
     */
    public static final String DEFAULT_KEY_STORE_FILE = "${java.home}${/}.keystore";

    private CallbackHandler handler;

    private Subject subject;

    private boolean debug = false;

    private Set principals = new HashSet();

    private Set publicCreds = new HashSet();

    private Set privateCreds = new HashSet();

    protected KeyStore keyStore;

    /**
     * Creates a new KeyStoreLoginModule.
     */
    public KeyStoreLoginModule() {
    }

    /**
     * Initializes the login module.
     *
     * @see LoginModule#initialize(Subject, CallbackHandler, Map, Map)
     */
    public void initialize(Subject sub, CallbackHandler callbackHandler,
            Map mapSharedState, Map mapOptions) {

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String) mapOptions.get("debug"));

        if (sub == null)
            throw new IllegalArgumentException("Subject is not specified");

        if (callbackHandler == null)
            throw new IllegalArgumentException(
            "CallbackHander is not specified");

        // Get the key store path
        String sKeyStorePath = (String) mapOptions
            .get(KEY_STORE_FILE_PROPERTY_NAME);

        // If there is no key store path, the default one is the .keystore
        // file in the java home directory
        if (sKeyStorePath == null) {
            sKeyStorePath = DEFAULT_KEY_STORE_FILE;
        }

        // Replace the system enviroment variable
        sKeyStorePath = ObjectGridUtil.replaceVar(sKeyStorePath);

        File fileKeyStore = new File(sKeyStorePath);

        try {
            KeyStore store = KeyStore.getInstance("JKS");
            store.load(new FileInputStream(fileKeyStore), null);

            // Save the key store
            keyStore = store;

            if (debug) {
                System.out.println("[KeyStoreLoginModule] initialize: Successfully loaded key store");
            }
        }
        catch (Exception e) {
            ObjectGridRuntimeException re = new ObjectGridRuntimeException(
                    "Failed to load keystore: " + fileKeyStore.getAbsolutePath());
            re.initCause(e);
            if (debug) {
                System.out.println("[KeyStoreLoginModule] initialize: Key store loading failed with exception "
                        + e.getMessage());
            }
        }

        this.subject = sub;
        this.handler = callbackHandler;
    }

    /**
     * Authenticates a user based on the keystore file.
     *
     * @see LoginModule#login()
     */
    public boolean login() throws LoginException {

        if (debug) {
            System.out.println("[KeyStoreLoginModule] login: entry");
        }

        String name = null;
        char pwd[] = null;

        if (keyStore == null || subject == null || handler == null) {
            throw new LoginException("Module initialization failed");
        }

        NameCallback nameCallback = new NameCallback("Username:");
        PasswordCallback pwdCallback = new PasswordCallback("Password:", false);

        try {
            handler.handle(new Callback[] { nameCallback, pwdCallback });
        }
        catch (Exception e) {
            throw new LoginException("Callback failed: " + e);
        }

        name = nameCallback.getName();
        char[] tempPwd = pwdCallback.getPassword();

        if (tempPwd == null) {
            // treat a NULL password as an empty password
            tempPwd = new char[0];
        }
        pwd = new char[tempPwd.length];
        System.arraycopy(tempPwd, 0, pwd, 0, tempPwd.length);

        pwdCallback.clearPassword();

        if (debug) {
            System.out.println("[KeyStoreLoginModule] login: "
                    + "user entered user name: " + name);
        }

        // Validate the user name and password
        try {
            validate(name, pwd);
        }
        catch (SecurityException se) {
            principals.clear();
            publicCreds.clear();
            privateCreds.clear();
            LoginException le = new LoginException(
            "Exception encountered during login");
            le.initCause(se);

            throw le;
        }

        if (debug) {
            System.out.println("[KeyStoreLoginModule] login: exit");
        }
        return true;
    }

    /**
     * Indicates the user is accepted.
     * <p>
     * This method is called only if the user is authenticated by all modules in
     * the login configuration file. The principal objects will be added to the
```

```
 * stored subject.
 *
 * @return false if for some reason the principals cannot be added; true
 *         otherwise
 *
 * @exception LoginException
 *                LoginException is thrown if the subject is readonly or if
 *                any unrecoverable exceptions is encountered.
 *
 * @see LoginModule#commit()
 */
public boolean commit() throws LoginException {
    if (debug) {
        System.out.println("[KeyStoreLoginModule] commit: entry");
    }

    if (principals.isEmpty()) {
        throw new IllegalStateException("Commit is called out of sequence");
    }

    if (subject.isReadOnly()) {
        throw new LoginException("Subject is Readonly");
    }

    subject.getPrincipals().addAll(principals);
    subject.getPublicCredentials().addAll(publicCreds);
    subject.getPrivateCredentials().addAll(privateCreds);

    principals.clear();
    publicCreds.clear();
    privateCreds.clear();

    if (debug) {
        System.out.println("[KeyStoreLoginModule] commit: exit");
    }
    return true;
}

/**
 * Indicates the user is not accepted
 *
 * @see LoginModule#abort()
 */
public boolean abort() throws LoginException {
    boolean b = logout();
    return b;
}

/**
 * Logs the user out. Clear all the maps.
 *
 * @see LoginModule#logout()
 */
public boolean logout() throws LoginException {


    // Clear the instance variables
    principals.clear();
    publicCreds.clear();
    privateCreds.clear();

    // clear maps in the subject
    if (!subject.isReadOnly()) {
        if (subject.getPrincipals() != null) {
            subject.getPrincipals().clear();
        }

        if (subject.getPublicCredentials() != null) {
            subject.getPublicCredentials().clear();
        }

        if (subject.getPrivateCredentials() != null) {
            subject.getPrivateCredentials().clear();
        }
    }
    return true;
}

/**
 * Validates the user name and password based on the keystore.
 *
 * @param userName user name
 * @param password password
 * @throws SecurityException if any exceptions encountered
 */
private void validate(String userName, char password[])
    throws SecurityException {

    PrivateKey privateKey = null;

    // Get the private key from the keystore
```

```
        try {
            privateKey = (PrivateKey) keyStore.getKey(userName, password);
        }
        catch (NoSuchAlgorithmException nsae) {
            SecurityException se = new SecurityException();
            se.initCause(nsae);
            throw se;
        }
        catch (KeyStoreException kse) {
            SecurityException se = new SecurityException();
            se.initCause(kse);
            throw se;
        }
        catch (UnrecoverableKeyException uke) {
            SecurityException se = new SecurityException();
            se.initCause(uke);
            throw se;
        }

        if (privateKey == null) {
            throw new SecurityException("Invalid name: " + userName);
        }

        // Check the certificats
        Certificate certs[] = null;
        try {
            certs = keyStore.getCertificateChain(userName);
        }
        catch (KeyStoreException kse) {
            SecurityException se = new SecurityException();
            se.initCause(kse);
            throw se;
        }

        if (debug) {
            System.out.println("  Print out the certificates:");
            for (int i = 0; i < certs.length; i++) {
                System.out.println("  certificate " + i);
                System.out.println("    " + certs[i]);
            }
        }

        if (certs != null && certs.length > 0) {

            // If the first certificate is an X509Certificate
            if (certs[0] instanceof X509Certificate) {
                try {
                    // Get the first certificate which represents the user
                    X509Certificate certX509 = (X509Certificate) certs[0];

                    // Create a principal
                    X500Principal principal = new X500Principal(certX509
                            .getIssuerDN()
                            .getName());
                    principals.add(principal);

                    if (debug) {
                        System.out.println("  Principal added: " + principal);
                    }
                    // Create the certification path object and add it to the
                    // public credential set
                    CertificateFactory factory = CertificateFactory
                        .getInstance("X.509");
                    java.security.cert.CertPath certPath = factory
                        .generateCertPath(Arrays.asList(certs));
                    publicCreds.add(certPath);

                    // Add the private credential to the private credential set
                    privateCreds.add(new X500PrivateCredential(certX509,
                            privateKey, userName));

                }
                catch (CertificateException ce) {
                    SecurityException se = new SecurityException();
                    se.initCause(ce);
                    throw se;
                }
            }
            else {
                // The first certificate is not an X509Certificate
                // We just add the certificate to the public credential set
                // and the private key to the private credential set.
                publicCreds.add(certs[0]);
                privateCreds.add(privateKey);
            }
        }
    }
}
```

Using the LDAP authenticator plug-in

You are provided with the com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticator default implementation to handle the user name and password authentication to an LDAP server. This implementation uses the LDAPLogin login module to log the user into a Lightweight Directory Access Protocol (LDAP) server.The following snippet demonstrates how the authenticate method is implemented:

```
/**
* @see com.ibm.ws.objectgrid.security.plugins.Authenticator#
* authenticate(LDAPLogin)
*/
public Subject authenticate(Credential credential) throws
InvalidCredentialException, ExpiredCredentialException {

    UserPasswordCredential cred = (UserPasswordCredential) credential;
    LoginContext lc = null;
    try {
        lc = new LoginContext("LDAPLogin",
            new UserPasswordCallbackHandlerImpl(cred.getUserName(),
            cred.getPassword().toCharArray()));

        lc.login();

        Subject subject = lc.getSubject();

        return subject;
    }
    catch (LoginException le) {
        throw new InvalidCredentialException(le);
    }
    catch (IllegalArgumentException ile) {
        throw new InvalidCredentialException(ile);
    }
}
```

Also, eXtreme Scale ships a login module com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule for this purpose. You must provide the following two options in the JAAS login configuration file.

- providerURL: The LDAP server provider URL
- factoryClass: The LDAP context factory implementation class

The LDAPLoginModule module calls the com.ibm.websphere.objectgrid.security.plugins.builtins. LDAPAuthenticationHelper.authenticate method. The following code snippet shows how you can implement the authenticate method of the LDAPAuthenticationHelper.

```
/**
* Authenticate the user to the LDAP directory.
* @param user the user ID, e.g., uid=xxxxxx,c=us,ou=bluepages,o=ibm.com
* @param pwd the password
*
* @throws NamingException
*/
public String[] authenticate(String user, String pwd)
throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, factoryClass);
    env.put(Context.PROVIDER_URL, providerURL);
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, pwd);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");

    InitialContext initialContext = new InitialContext(env);

    // Look up for the user
    DirContext dirCtx = (DirContext) initialContext.lookup(user);

    String uid = null;
    int iComma = user.indexOf(",");
```

```
        int iEqual = user.indexOf("=");
        if (iComma > 0 && iComma > 0) {
            uid = user.substring(iEqual + 1, iComma);
        }
        else {
            uid = user;
        }

        Attributes attributes = dirCtx.getAttributes("");

        // Check the UID
        String thisUID = (String) (attributes.get(UID).get());

        String thisDept = (String) (attributes.get(HR_DEPT).get());

        if (thisUID.equals(uid)) {
            return new String[] { thisUID, thisDept };
        }
        else {
            return null;
        }
    }
```

If authentication succeeds, the ID and password are considered valid. Then the login module gets the ID information and department information from this authenticate method. The login module creates two principals: SimpleUserPrincipal and SimpleDeptPrincipal. You can use the authenticated subject for group authorization (in this case, the department is a group) and individual authorization.

The following example shows a login module configuration that is used to log in to the LDAP server:

```
LDAPLogin { com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule required
    providerURL="ldap://directory.acme.com:389/"
    factoryClass="com.sun.jndi.ldap.LdapCtxFactory";
};
```

In the previous configuration, the LDAP server points to the `ldap://directory.acme.com:389/server`. Change this setting to your LDAP server. This login module uses the provided ID and password to connect to the LDAP server. This implementation is for testing purposes only.

Using the WebSphere Application Server authenticator plug-in

Also, eXtreme Scale provides the com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator built-in implementation to use the WebSphere Application Server security infrastructure. This built-in implementation can be used when the following conditions are true.

1. WebSphere Application Server global security is turned on.
2. All eXtreme Scale clients and servers are launched in WebSphere Application Server JVMs.
3. These application servers are in the same security domain.
4. The eXtreme Scale client is already authenticated in WebSphere Application Server.

The client can use the com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator class to generate a credential. The server uses this Authenticator implementation class to authenticate the credential. If the token is authenticated successfully, a Subject object returns.

This scenario takes advantage of the fact that the client has already been authenticated. Because the application servers that have the servers are in the same security domain as the application servers that house the clients, the security tokens can be propagated from the client to the server so that the same user registry does not need to be authenticated again.

Using the Tivoli® Access Manager authenticator plug-in

Tivoli Access Manager is used widely as a security server. You can also implement Authenticator using the Tivoli Access Manager's provided login modules.

To authenticate a user for Tivoli Access Manager, apply the the com.tivoli.mts.PDLoginModule login module, which requires that the calling application provide the following information:

1. A principal name, specified as either a short name or an X.500 name (DN)
2. A password

The login module authenticates the principal and returns the Tivoli Access Manager credential. The login module expects the calling application to provide the following information:

1. The user name, through a javax.security.auth.callback.NameCallback object.
2. The password, through a javax.security.auth.callback.PasswordCallback object.

When the Tivoli Access Manager credential is successfully retrieved, the JAAS LoginModule creates a Subject and a PDPrincipal. No built-in for Tivoli Access Manager authentication is provided, because it is just with the PDLoginModule module. See the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

Connecting to WebSphere eXtreme Scale securely

To connect an eXtreme Scale client to a server securely, you can use any connect method in the ObjectGridManager interface which takes a ClientSecurityConfiguration object. The following is a brief example.

```
public ClientClusterContext connect(String catalogServerAddresses,
   ClientSecurityConfiguration securityProps,
     URL overRideObjectGridXml) throws ConnectException;
```

This method takes a parameter of the ClientSecurityConfiguration type, which is an interface representing a client security configuration. You can use com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory public API to create an instance with default values, or you can create an instance by passing the WebSphere eXtreme Scale client property file. This file contains the following properties that are related to authentication. The value marked with a plus sign (+) is the default.

- `securityEnabled (true, false+):` This property indicates if security is enabled. When a client connects to a server, the securityEnabled value on the client and server side must be both `true` or both `false`. For example, if the connected server security is enabled, the client has to set this property to true to connect to the server.
- authenticationRetryCount (an integer value, 0+): This property determines how many retries are attempted for login when a credential is expired. If the value is 0, no retries are attempted. The authentication retry only applies to the case when the credential is expired. If the credential is not valid, there is no retry. Your application is responsible for retrying the operation.

After you create a com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration object, set the credentialGenerator object on the client using the following method:

```
/**
* Set the {@link CredentialGenerator} object for this client.
* @param generator the CredentialGenerator object associated with this client
*/
void setCredentialGenerator(CredentialGenerator generator);
```

You can set the CredentialGenerator object in the WebSphere eXtreme Scale client property file too, as follows.

- credentialGeneratorClass: The class implementation name for the CredentialGenerator object. It must have a default constructor.
- credentialGeneratorProps: The properties for the CredentialGenerator class. If the value is not null, it is set to the constructed CredentialGenerator object using the setProperties(String) method.

Here is a sample to instantiate a ClientSecurityConfiguration and then use it to connect to the server.

```
/**
* Get a secure ClientClusterContext
* @return a secure ClientClusterContext object
*/
protected ClientClusterContext connect() throws ConnectException {
ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
.getClientSecurityConfiguration("/properties/security.ogclient.props");

UserPasswordCredentialGenerator gen= new
UserPasswordCredentialGenerator("manager", "manager1");

csConfig.setCredentialGenerator(gen);

return objectGridManager.connect(csConfig, null);
}
```

When the connect is called, the WebSphere eXtreme Scale client calls the CredentialGenerator.getCredential method to get the client credential. This credential is sent along with the connect request to the server for authentication.

Using a different CredentialGenerator instance per session

In some cases, a WebSphere eXtreme Scale client represents just one client identity, but in others, it might represent multiple identities. Here is one scenario for the latter case: An WebSphere eXtreme Scale client is created and shared in a Web server. All servlets in this Web server use this one WebSphere eXtreme Scale client. Because every servlet represents a different Web client, use different credentials when sending requests to WebSphere eXtreme Scale servers.

WebSphere eXtreme Scale provides for changing the credential on the session level. Every session can uses a different CredentialGenerator object. Therefore, the previous scenarios can be implemented by letting the servlet get a session with a different CredentialGenerator object. The following example illustrates the ObjectGrid.getSession(CredentialGenerator) method in the ObjectGridManager interface.

```
/**
     * Get a session using a <code>CredentialGenerator</code>.
     * <p>
     * This method can only be called by the ObjectGrid client in an ObjectGrid
     * client server environment. If ObjectGrid is used in a local model, that is,
     * within the same JVM with no client or server existing, <code>getSession(Subject)</code>
     * or the <code>SubjectSource</code> plugin should be used to secure the ObjectGrid.
     *
     * <p>If the <code>initialize()</code> method has not been invoked prior to
     * the first <code>getSession</code> invocation, an implicit initialization
     * will occur.  This ensures that all of the configuration is complete
     * before any runtime usage is required.</p>
     *
     * @param credGen A <code>CredentialGenerator</code> for generating a credential
     *               for the session returned.
     *
     * @return An instance of <code>Session</code>
     *
     * @throws ObjectGridException if an error occurs during processing
     * @throws TransactionCallbackException if the <code>TransactionCallback</code>
     *        throws an exception
     * @throws IllegalStateException if this method is called after the
     *        <code>destroy()</code> method is called.
     *
     * @see #destroy()
     * @see #initialize()
     * @see CredentialGenerator
     * @see Session
     * @since WAS XD 6.0.1
 */
Session getSession(CredentialGenerator credGen) throws
ObjectGridException, TransactionCallbackException;
```

The following is an example:

```
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

CredentialGenerator credGenManager = new UserPasswordCredentialGenerator("manager", "xxxxxx");
CredentialGenerator credGenEmployee = new UserPasswordCredentialGenerator("employee", "xxxxxx");

ObjectGrid og = ogManager.getObjectGrid(ctx, "accounting");

// Get a session with CredentialGenerator;
Session session = og.getSession(credGenManager );

// Get the employee map
ObjectMap om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec1 = map.get("xxxxxx");

session.commit();

// Get another session with a different CredentialGenerator;
session = og.getSession(credGenEmployee );

// Get the employee map
om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec2 = map.get("xxxxx");

session.commit();
```

If you use the ObjectGird.getSession method to get a Session object, the session uses the CredentialGenerator object set on the ClientConfigurationSecurity object. The ObjectGrid.getSession(CredentialGenerator) method overrides the CredentialGenerator set in the ClientSecurityConfiguration object.

If you can reuse the Session object, a performance gain results. However, calling the ObjectGrid.getSession(CredentialGenerator) method is not very expensive. The major overhead is the increased object garbage collection time. Make sure that you release the references after you are done with the Session objects. Generally, if your Session object can share the identity, try to reuse the Session object. If not, use the ObjectGrid.getSession(CredentialGenerator) method.

Client authorization programming

WebSphere eXtreme Scale supports Java Authentication and Authorization Service (JAAS) authorization out-of-the-box and also supports custom Authorization using the ObjectGridAuthorization interface.

The ObjectGridAuthorization plug-in is used to authorize ObjectGrid, ObjectMap and JavaMap accesses to the Principals represented by a Subject object in a custom way. A typical implementation of this plug-in is to retrieve the Principals from the Subject object, and then check whether or not the specified permissions are granted to the Principals.

A permission passed to the checkPermission(Subject, Permission) method can be one of the following permissions:

1. MapPermission
2. ObjectGridPermission
3. ServerMapPermission
4. AgentPermission

Refer to ObjectGridAuthorization API Documentation for more details.

MapPermission

The com.ibm.websphere.objectgrid.security.MapPermission public class represents permissions to the ObjectGrid resources, specifically the methods of ObjectMap or JavaMap interfaces. WebSphere eXtreme Scale defines the following permission strings to access the methods of ObjectMap and JavaMap:

1. **read**: Permission to read the data from the map. The integer constant is defined as `MapPermission.READ`.
2. **write**: Permission to update the data in the map. The integer constant is defined as `MapPermission.WRITE`.
3. **insert**: Permission to insert the data into the map. The integer constant is defined as `MapPermission.INSERT`.
4. **remove**: Permission to remove the data from the map. The integer constant is defined as `MapPermission.REMOVE`.
5. **invalidate**: Permission to invalidate the data from the map. The integer constant is defined as `MapPermission.INVALIDATE`.
6. **all**: All above permissions: read, write, insert, remote, and invalidate. The integer constant is defined as `MapPermission.ALL`.

Refer to MapPermission API Documentation for more details.

You can construct a MapPermission object by passing the fully qualified ObjectGrid map name (in format [ObjectGrid_name].[ObjectMap_name]) and the permission string or integer value. A permission string can be a comma-delimited string of the previous permission strings such as read, insert, or it can be all. A permission integer value can be any previously mentioned permission integer constants or a mathematical value of several integer permission constants, such as MapPermission.READ|MapPermission.WRITE.

The authorization occurs when an ObjectMap or JavaMap method is called. The eXtreme Scale runtime checks different permissions for different methods. If the required permissions are not granted to the client, an AccessControlException results.

Table 12. List of methods and the required MapPermission

Permission	ObjectMap/JavaMap
read	boolean containsKey(Object)
	boolean equals(Object)
	Object get(Object)
	Object get(Object, Serializable)
	List getAll(List)
	List getAll(List keyList, Serializable)
	List getAllForUpdate(List)
	List getAllForUpdate(List, Serializable)
	Object getForUpdate(Object)
	Object getForUpdate(Object, Serializable)
	public Object getNextKey(long)
write	Object put(Object key, Object value)
	void put(Object, Object, Serializable)
	void putAll(Map)
	void putAll(Map, Serializable)
	void update(Object, Object)
	void update(Object, Object, Serializable)
insert	public void insert (Object, Object)
	void insert(Object, Object, Serializable)
remove	Object remove (Object)
	void removeAll(Collection)
	void clear()
invalidate	public void invalidate (Object, boolean)
	void invalidateAll(Collection, boolean)
	void invalidateUsingKeyword(Serializable)
	int setTimeToLive(int)

Authorization is based solely on which method is used, rather than what the method really does. For example, a put method can insert or update a record based on whether the record exists. However, the insert or update cases are not distinguished.

An operation type can be achieved by combinations of other types. For example, an update can be achieved by a remove and then an insert. Consider these combinations when designing your authorization policies.

ObjectGridPermission

A com.ibm.websphere.objectgrid.security.ObjectGridPermission represents permissions to the ObjectGrid:

- Query: permission to create an object query or entity query. The integer constant is defined as ObjectGridPermission.QUERY.

- Dynamic map: permission to create a dynamic map based on the map template. The integer constant is defined as ObjectGridPermission.DYNAMIC_MAP.

Refer to ObjectGridPermission API Documentation for more details.

The following table summarizes the methods and the required ObjectGridPermission:

Table 13. List of methods and the required ObjectGridPermission

Permission action	Methods
query	com.ibm.websphere.objectgrid.Session.createObjectQuery(String)
query	com.ibm.websphere.objectgrid.em.EntityManager.createQuery(String)
dynamicmap	com.ibm.websphere.objectgrid.Session.getMap(String)

ServerMapPermission

An ServerMapPermission represents permissions to an ObjectMap hosted in a server. The name of the permission is the full name of the ObjectGrid map name. It has three actions:

1. replicate: permission to replicate a server map to near cache.

2. dynamicIndex: permission for a client to create or remove a dynamic index on a server

Refer to ServerMapPermission API documentation for more details. The detailed methods, which require different ServerMapPermission, are listed in the following table:

Table 14. Permissions to a server-hosted ObjectMap

Permission action	Methods
replicate	com.ibm.websphere.objectgrid.ClientReplicableMap.enableClientReplication(Mode, int[], ReplicationMapListener)
dynamicIndex	com.ibm.websphere.objectgrid.BackingMap.createDynamicIndex(String, boolean, String, DynamicIndexCallback)
dynamicIndex	com.ibm.websphere.objectgrid.BackingMap.removeDynamicIndex(String)

AgentPermission

An AgentPermission represents permissions to the datagrid agents. The name of the permission is the full name of the ObjectGrid map, and the action is a comma-delimited string of agent implementation class names or package names.

Refer to AgentPermission API Documentation for more information.

The following methods in the class com.ibm.websphere.objectgrid.datagrid.AgentManager require AgentPermission.

```
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
```

Authorization mechanisms

WebSphere eXtreme Scale supports two kinds of authorization mechanisms: Java Authentication and Authorization Service (JAAS) authorization and custom authorization. These mechanisms apply to all authorizations. JAAS authorization augments the Java security policies with user-centric access controls. Permissions can be granted based not just on what code is running, but also on who is running it. JAAS authorization is part of the SDK Version 1.4 and later.

Additionally, WebSphere eXtreme Scale also supports custom authorization with the following plug-in:

- ObjectGridAuthorization: custom way to authorize access to all artifacts.

You can implement your own authorization mechanism if you do not want to use JAAS authorization. By using a custom authorization mechanism, you can use the policy database, policy server, or Tivoli Access Manager to manage the authorizations.

You can configure the authorization mechanism in two ways:

1. *XML Configuration:* You can use the ObjectGrid XML file to define an ObjectGrid and set the authorization mechanism to either AUTHORIZATION_MECHANISM_JAAS or AUTHORIZATION_MECHANISM_CUSTOM. Here is the secure-objectgrid-definition.xml file that is used in the enterprise application ObjectGridSample:

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
  <bean id="TransactionCallback"
classname="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
...
</objectGrids>
```

2. *Programmatic Configuration:* If you want to create an ObjectGrid using method ObjectGrid.setAuthorizationMechanism(int), you can call the following method to set the authorization mechanism. Calling this method applies only to the local WebSphere eXtreme Scale programming model when you directly instantiate the ObjectGrid instance:

```
/**
 * Set the authorization Mechanism. The default is
 * com.ibm.websphere.objectgrid.security.SecurityConstants.
 * AUTHORIZATION_MECHANISM_JAAS.
 * @param authMechanism the map authorization mechanism
*/
void setAuthorizationMechanism(int authMechanism);
```

JAAS authorization

A javax.security.auth.Subject object represents an authenticated user. A Subject is comprised of a set of principals, and each Principal represents an identity for that user. For example, a Subject can have a name principal, for example, Joe Smith, and a group principal, for example, manager.

Using the JAAS authorization policy, permissions can be granted to specific Principals. WebSphere eXtreme Scale associates the Subject with the current access control context. For each call to the ObjectMap or Javamap method, the Java runtime automatically determines if the policy grants the required permission only to a specific Principal and if so, the operation is allowed only if the Subject associated with the access control context contains the designated Principal.

You must be familiar with the policy syntax of the policy file. For detailed description of JAAS authorization, refer to the JAAS Reference Guide.

WebSphere eXtreme Scale has a special code base that is used for checking the JAAS authorization to the ObjectMap and JavaMap method calls. This special code base is `http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction`. Use this code base when granting ObjectMap or JavaMap permissions to principals. This special code was created because the Java archive (JAR) file for eXtreme Scale is granted with all permissions.

The template of the policy to grant the MapPermission permission is:

```
grant codeBase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
   <Principal field(s)>{
    permission com.ibm.websphere.objectgrid.security.MapPermission
              "[ObjectGrid_name].[ObjectMap_name]", "action";
    ....
    permission com.ibm.websphere.objectgrid.security.MapPermission
              "[ObjectGrid_name].[ObjectMap_name]", "action";
  };
```

A Principal field looks like the following example:

```
principal Principal_class "principal_name"
```

In this policy, only insert and read permissions are granted to these four maps to a certain principal. The other policy file, fullAccessAuth.policy, grants all permissions to these maps to a principal. Before running the application, change the principal_name and principal class to appropriate values. The value of the principal_name depends on the user registry. For example, if local OS is used as user registry, the machine name is MACH1, the user ID is user1, and the principal_name is MACH1/user1.

The JAAS authorization policy can be put directly into the Java policy file, or it can be put in a separate JAAS authorization file and then set by using the

```
-Djava.security.auth.policy=file:[JAAS_AUTH_POLICY_FILE]
```

JVM argument or by using the

```
-Dauth.policy.url.x=file:[JAAS_AUTH_POLICY_FILE]
```

property in the java.security file.

Custom ObjectGrid authorization

ObjectGridAuthorization plug-in is used to authorize ObjectGrid, ObjectMap and JavaMap accesses to the Principals represented by a Subject object in a custom way. A typical implementation of this plug-in is to retrieve the Principals from the Subject object, and then check whether or not the specified permissions are granted to the Principals.

A permission passed to the checkPermission(Subject, Permission) method could be one of the following:

1. MapPermission

2. ObjectGridPermission

3. AgentPermission

4. ServerMapPermission

Refer to ObjectGridAuthorization API documentation for more details.

The ObjectGridAuthorization plug-in can be configured in the following ways:

1. *XML Configuration:*You can use the ObjectGrid XML file to define an ObjectAuthorization plug-in. Here is an example:

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_CUSTOM">
...
  <bean id="ObjectGridAuthorization"
className="com.acme.ObjectGridAuthorizationImpl" />
</objectGrids>
```

2. *Programmatic Configuration:* If you want to create an ObjectGrid using the API method ObjectGrid.setObjectGridAuthorization(ObjectGridAuthorization), you can call the following method to set the authorization plug-in. This method only applies to the local eXtreme Scale programming model when you directly instantiate the ObjectGrid instance.

```
/**
     * Sets the <code>ObjectGridAuthorization</code> for this ObjectGrid instance.
     * <p>
     * Passing <code>null</code> to this method removes a previously set
     * <code>ObjectGridAuthorization</code> object from an earlier invocation of this method
     * and indicates that this <code>ObjectGrid</code> is not associated with a
     * <code>ObjectGridAuthorization</code> object.
     * <p>
     * This method should only be used when ObjectGrid security is enabled. If
     * the ObjectGrid security is disabled, the provided <code>ObjectGridAuthorization</code> object
     * will not be used.
     * <p>
     * A <code>ObjectGridAuthorization</code> plugin can be used to authorize
     * access to the ObjectGrid and maps. Please refer to <code>ObjectGridAuthorization</code> for more details.
     *
     * <p>
     * As of XD 6.1, the <code>setMapAuthorization</code> is deprecated and
     * <code>setObjectGridAuthorization</code> is recommended for use. However,
     * if both <code>MapAuthorization</code> plugin and <code>ObjectGridAuthorization</code> plugin
     * are used, ObjectGrid will use the provided <code>MapAuthorization</code> to authorize map accesses,
     * even though it is deprecated.
     * <p>
     * Note, to avoid an <code>IllegalStateException</code>, this method must be
     * called prior to the <code>initialize()</code> method.  Also, keep in mind
     * that the <code>getSession</code> methods implicitly call the
     * <code>initialize()</code> method if it has yet to be called by the
     * application.
     *
     * @param ogAuthorization the <code>ObjectGridAuthorization</code> plugin
     *
     * @throws IllegalStateException if this method is called after the
     *         <code>initialize()</code> method is called.
     *
     * @see #initialize()
     * @see ObjectGridAuthorization
     * @since WAS XD 6.1
     */
    void setObjectGridAuthorization(ObjectGridAuthorization ogAuthorization);
```

Implementing ObjectGridAuthorization

The boolean checkPermission(Subject subject, Permission permission) method of the ObjectGridAuthorization interface is called by theWebSphere eXtreme Scale run time to check whether the passed-in subject object has the passed-in permission. The implementation of the ObjectGridAuthorization interface returns true if the object has the permission, and false if not.

A typical implementation of this plug-in is to retrieve the principals from the Subject object and check whether the specified permissions are granted to the principals by consulting specific policies. These policies are defined by users. For example, the policies can be defined in a database, a plain file, or a Tivoli Access Manager policy server.

For example, we can use Tivoli Access Manager policy server to manage the authorization policy and use its API to authorize the access. For how to use Tivoli Access Manager Authorization APIs, refer to the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

This sample implementation has the following assumptions:

1. Only check authorization for MapPermission. For other permissions, always return true.
2. The Subject object contains a com.tivoli.mts.PDPrincipal principal.
3. The Tivoli Access Manager policy server has defined the following permissions for the ObjectMap or JavaMap name object. The object that is defined in the policy server must have the same name as the ObjectMap or JavaMap name in the format of [ObjectGrid_name].[ObjectMap_name]. The permission is the first character of the permission strings that are defined in the MapPermission permission. For example, the permission "r" that is defined in the policy server represents the read permission to the ObjectMap map.

The following code snippet demonstrates how to implement the checkPermission method:

```
/**
* @see com.ibm.websphere.objectgrid.security.plugins.
*   MapAuthorization#checkPermission
* (javax.security.auth.Subject, com.ibm.websphere.objectgrid.security.
*   MapPermission)
*/
public boolean checkPermission(final Subject subject,
 Permission p) {

  // For non-MapPermission, we always authorize.
  if (!(p instanceof MapPermission)){
    return true;
  }

  MapPermission permission = (MapPermission) p;

  String[] str = permission.getParsedNames();

  StringBuffer pdPermissionStr = new StringBuffer(5);
  for (int i=0; i<str.length; i++) {
    pdPermissionStr.append(str[i].substring(0,1));
  }

  PDPermission pdPerm = new PDPermission(permission.getName(),
  pdPermissionStr.toString());

  Set principals = subject.getPrincipals();

  Iterator iter= principals.iterator();
  while(iter.hasNext()) {
    try {
      PDPrincipal principal = (PDPrincipal) iter.next();
      if (principal.implies(pdPerm)) {
        return true;
      }
    }
    catch (ClassCastException cce) {
      // Handle exception
    }
  }
  return false;
}
```

Data grid authentication

You can use the secure token manager plug-in to enable server-to-server authentication, which requires you to implement the SecureTokenManager interface.

The generateToken(Object) method takes an object protect, and then generates a token that cannot be understood by others. The verifyTokens(byte[]) method does the reverse process: it converts the token back to the original object.

A simple SecureTokenManager implementation uses a simple encoding algorithm, such as a XOR algorithm, to encode the object in serialized form and then use corresponding decoding algorithm to decode the token. This implementation is not secure and is easy to break.

WebSphere eXtreme Scale default implementation

WebSphere eXtreme Scale provides an immediately available implementation for this interface. This default implementation uses a key pair to sign and verify the signature, and uses a secret key to encrypt the content. Every server has a JCKES type keystore to store the key pair, a private key and public key, and a secret key. The keystore has to be the JCKES type to store secret keys. These keys are used to encrypt and sign or verify the secret string on the sending end. Also, the token is associated with an expiration time. On the receiving end, the data is verified, decrypted, and compared to the receiver secret string. Secure Sockets Layer (SSL) communication protocols are not required between a pair of servers for authentication because the private keys and public keys serve the same purpose. However, if server communication is not encrypted, the data can be stolen by looking at the communication. Because the token expires soon, the replay attack threat is minimized. This possibility is significantly decreased if all servers are deployed behind a firewall.

The disadvantage of this approach is that the WebSphere eXtreme Scale administrators have to generate keys and transport them to all servers, which can cause security breach during transportation.

Local security

WebSphere eXtreme Scale provides several security endpoints to allow you to integrate custom mechanisms. In the local programming model, the main security function is authorization, and has no authentication support . You must authenticate outside of WebSphere Application Server. However, there are provided plug-ins to obtain and validate Subject objects.

Enabling security

The following list provides the two ways in which local security is enabled:

- **XML Configuration** You can use the ObjectGrid XML file to define an ObjectGrid and enable the security for that ObjectGrid. The following file is the secure-objectgrid-definition.xml file that is used in the ObjectGridSample enterprise application sample. In this XML file, security is enabled by setting the securityEnabled attribute to true.

```
<objectGrids>
    <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
        authorizationMechanism="AUTHORIZATION_MECHANISM_JASS">
    ...
</objectGrids>
```

- **Programming** If you want to create an ObjectGrid using the API method ObjectGrid.setSecurityEnabled(), call the following method on the ObjectGrid interface to enable security.

```
/**
 * Enable the ObjectGrid security
 */
void setSecurityEnabled();
```

Authentication

In the local programming model, eXtreme Scale does not provide any authentication mechanism, but relies on the environment, either application servers or applications, for authentication. When eXtreme Scale is used in WebSphere Application Server or WebSphere Extended Deployment, applications can use the WebSphere Application Server security authentication mechanism. When eXtreme Scale is running in a Java 2 Platform, Standard Edition (J2SE) environment, the application has to manage authentications with Java Authentication and Authorization Service (JAAS) authentication or other authentication mechanisms. For more information about using JAAS authentication, see the JAAS reference guide. The contract between an application and an ObjectGrid instance is the javax.security.auth.Subject object. After the client is authenticated by the application server or the application, the application can retrieve the authenticated javax.security.auth.Subject object and use this Subject object to get a session from the ObjectGrid instance by calling the ObjectGrid.getSession(Subject) method. This Subject object is used to authorize accesses to the map data. This contract is called a subject passing mechanism. The following example illustrates the ObjectGrid.getSession(Subject) API.

```
/**
 * This API allows the cache to use a specific subject rather than the one
 * configured on the ObjectGrid to get a session.
 * @param subject
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws InvalidSubjectException the subject passed in is not valid based
 * on the SubjectValidation mechanism.
 */
public Session getSession(Subject subject)
throws ObjectGridException, TransactionCallbackException, InvalidSubjectException;
```

The ObjectGrid.getSession() method in the ObjectGrid interface can also be used to get a Session object:

```
/**
 * This method returns a Session object that can be used by a single thread at a time.
 * You cannot share this Session object between threads without placing a
 * critical section around it. While the core framework allows the object to move
 * between threads, the TransactionCallback and Loader might prevent this usage,
 * especially in J2EE environments. When security is enabled, this method uses the
 * SubjectSource to get a Subject object.
 *
 * If the initialize method has not been invoked prior to the first
 * getSession invocation, then an implicit initialization occurs.  This
 * initialization ensures that all of the configuration is complete before
 * any runtime usage is required.
 *
 * @see #initialize()
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws IllegalStateException if this method is called after the
 *         destroy() method is called.
 */
public Session getSession()
throws ObjectGridException, TransactionCallbackException;
```

As the API documentation specifies, when security is enabled, this method uses the SubjectSource plug-in to get a Subject object. The SubjectSource plug-in is one of the security plug-ins defined in eXtreme Scale to support propagating Subject objects. See Security-related plug-ins for more information. The getSession(Subject) method can be called on the local ObjectGrid instance only. If you call the getSession(Subject) method on a client side in a distributed eXtreme Scale configuration, an IllegalStateException results.

Security plug-ins

WebSphere eXtreme Scale provides two security plug-ins that are related to the subject passing mechanism: the SubjectSource and SubjectValidation plug-ins.

SubjectSource plug-in

The SubjectSource plug-in, represented by the com.ibm.websphere.objectgrid.security.plugins.SubjectSource interface, is a plug-in that is used to get a Subject object from an eXtreme Scale running environment. This environment can be an application using the ObjectGrid or an application server that hosts the application. Consider the SubjectSource plug-in an alternative to the subject passing mechanism. Using the subject passing mechanism, the application retrieves the Subject object and uses it to get the ObjectGrid session object. With the SubjectSource plug-in, the eXtreme Scale runtime retrieves the Subject object and uses it to get the session object. The subject passing mechanism gives the control of Subject objects to applications, while the SubjectSource plug-in mechanism frees applications from retrieving the Subject object. You can use the SubjectSource plug-in to get a Subject object that represents an eXtreme Scale client that is used for authorization. When the ObjectGrid.getSession method is called, the Subject getSubject throws an ObjectGridSecurityException if security is enabled. WebSphere eXtreme Scale provides a default implementation of this plug-in: com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectSourceImpl. This implementation can be used to retrieve a caller subject or a RunAs subject from the thread when an application is running in WebSphere Application Server. You can configure this class in your ObjectGrid descriptor XML file as the SubjectSource implementation class when using eXtreme Scale in WebSphere Application Server. The following code snippet shows the main flow of the WSSubjectSourceImpl.getSubject method.

```
Subject s = null;
try {
  if (finalType == RUN_AS_SUBJECT) {
    // get the RunAs subject
    s = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();
  }
  else if (finalType == CALLER_SUBJECT) {
    // get the callersubject
    s = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
  }
}
catch (WSSecurityException wse) {
  throw new ObjectGridSecurityException(wse);
}

return s;
```

For other details, refer to the API documentation for the SubjectSource plug-in and the WSSubjectSourceImpl implementation.

SubjectValidation plug-in

The SubjectValidation plug-in, which is represented by the com.ibm.websphere.objectgrid.security.plugins.SubjectValidation interface, is another security plug-in. The SubjectValidation plug-in can be used to validate that a javax.security.auth.Subject, either passed to the ObjectGrid or retrieved by the SubjectSource plug-in, is a valid Subject that has not been tampered with.

The SubjectValidation.validateSubject(Subject) method in the SubjectValidation interface takes a Subject object and returns a Subject object. Whether a Subject object is considered valid and which Subject object is returned are all up to your implementations. If the Subject object is not valid, an InvalidSubjectException results.

You can use this plug-in if you do not trust the Subject object that is passed to this method. This case is rare considering that you trust the application developers who develop the code to retrieve the Subject object.

An implementation of this plug-in needs support from the Subject object creator because only the creator knows if the Subject object has been tampered with. However, some subject creator might not know if the Subject has been tampered with. In this case, this plug-in is not useful.

WebSphere eXtreme Scale provides a default implementation of SubjectValidation: com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl. You can use this implementation to validate the WebSphere Application Server-authenticated subject. You can configure this class as the SubjectValidation implementation class when using eXtreme Scale in WebSphere Application Server. The WSSubjectValidationImpl implementation considers a Subject object valid only if the credential token that is associated with this Subject has not been tampered with. You can change other parts of the Subject object. The WSSubjectValidationImpl implementation asks WebSphere Application Server for the original Subject corresponding to the credential token and returns the original Subject object as the validated Subject object. Therefore, the changes made to the Subject contents other than the credential token have no effects. The following code snippet shows the basic flow of the WSSubjectValidationImpl.validateSubject(Subject).

```
// Create a LoginContext with scheme WSLogin and
// pass a Callback handler.
LoginContext lc = new LoginContext("WSLogin",
new WSCredTokenCallbackHandlerImpl(subject));

// When this method is called, the callback handler methods
// will be called to log the user in.
lc.login();

// Get the subject from the LoginContext
return lc.getSubject();
```

In the previous code snippet, a credential token callback handler object, WSCredTokenCallbackHandlerImpl, is created with the Subject object to validate. Then a LoginContext object is created with the login scheme WSLogin. When the lc.login method is called, WebSphere Application Server security retrieves the credential token from the Subject object and then returns the correspondent Subject as the validated Subject object.

For other details, refer to the Java APIs of SubjectValidation and WSSubjectValidationImpl implementation.

Plug-in configuration

You can configure the SubjectValidation plug-in and SubjectSource plug-in in two ways:

- **XML Configuration**You can use the ObjectGrid XML file to define an ObjectGrid and set these two plug-ins. Here is an example, in which the WSSubjectSourceImpl class is configured as the SubjectSource plug-in and the WSSubjectValidation class is configured as the SubjectValidation plug-in.

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
        <bean id="SubjectSource"
className="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectSourceImpl" />
    <bean id="SubjectValidation"
 className="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectValidationImpl" />
    <bean id="TransactionCallback"
className="com.ibm.websphere.samples.objectgrid.
 HeapTransactionCallback" />
...
</objectGrids>
```

- **Programming** If you want to create an ObjectGrid through APIs, you can call the following methods to set the SubjectSource or SubjectValidation plug-ins.

```
**
* Set the SubjectValidation plug-in for this ObjectGrid instance. A
* SubjectValidation plug-in can be used to validate the Subject object
* passed in as a valid Subject. Refer to {@link SubjectValidation}
* for more details.
* @param subjectValidation the SubjectValidation plug-in
*/
void setSubjectValidation(SubjectValidation subjectValidation);


/**
* Set the SubjectSource plug-in. A SubjectSource plug-in can be used
* to get a Subject object from the environment to represent the
* ObjectGrid client.
*
* @param source the SubjectSource plug-in
*/
void setSubjectSource(SubjectSource source);
```

Write your own JAAS authentication code

You can write you own Java Authentication and Authorization Service (JAAS) authentication code to handle the authentication. You need to write your own login modules and then configure the login modules for your authentication module.

The login module receives information about a user and authenticates the user. This information can be anything that can identify the user. For example, the information can be a user ID and password, client certificate, and so on. After receiving the information, the login module verifies that the information represents a valid subject and then creates a Subject object. Currently, several implementations of login modules are available to the public.

After a login module is written, configure this login module for the run time to use. You must configure a JAAS login module. This login module contains the login module and its authentication scheme. For example:

```
FileLogin
{
    com.acme.auth.FileLoginModule required
};
```

The authentication scheme is FileLogin and the login module is com.acme.auth.FileLoginModule. The required token indicates that the FileLoginModule module must validate this login or the entire scheme fails.

Setting the JAAS login module configuration file can be done in one of the following ways:

- Set the JAAS login module configuration file in the login.config.url property in the java.security file, for example:

  ```
  login.config.url.1=file:${java.home}/lib/security/file.login
  ```

- Set the JAAS login module configuration file from the command line by using the **-Djava.security.auth.login.config** Java virtual machine (JVM) arguments, for example, `-Djava.security.auth.login.config ==$JAVA_HOME/lib/security/file.login`

If your code is running in WebSphere Application Server, you must configure the JAAS login in the administrative console and store this login configuration in the application server configuration. See Login configuration for Java Authentication and Authorization Service for details.

Chapter 8. Performance considerations

To improve performance for your in-memory data grid or database processing space, you can investigate several considerations such as tuning your Java virtual machine settings and using the best practices for product features such as locking, serialization, and query performance.

JVM tuning

You must take into account several specific aspects of Java virtual machine (JVM) tuning for WebSphere eXtreme Scale best performance.

The recommendation is 1 to 2Gb heaps with a JVM per 4 cores. The heap sizes depend on the nature of the objects being stored in the servers, discussed later in this document.

Heap size and garbage collection recommendations

The optimum heap size number depends on three factors:
1. Number of live objects in the heap.
2. Complexity of live objects in the heap.
3. Number of available cores for the JVM.

For example, an application storing 10K byte arrays can run a much larger heap than an application using complex graphs of POJOs.

All modern JVMs today use parallel garbage collection algorithms, which means that using more cores can reduce pauses in garbage collection. So, 8-core boxes will be collected faster than a box with 4 cores.

Real memory usage versus heap specification

A 1Gb heap JVM uses approximately 1.3Gb of real memory. In our lab, we have been unable to run ten 1Gb JVMs on a box with 16Gb of RAM. Once the JVM heaps filled up to 800 plus MB, the box started paging.

Garbage collection

For IBM JVMs, use the avgoptpause collector for high update rate scenarios (100% of transactions modify entries). The gencon collector works much better than the avgoptpause collector for scenarios where data is updated relatively infrequently (10% of the time or less). Experiment with both collectors to see what works best in your scenario. If you see a performance problem, then run with verbose garbage collection turned on to check the percentage of the time that is being spent collecting garbage. Scenarios have occurred where 80% of the time is spent in garbage collection until tuning fixed the problem.

JVM performance

WebSphere eXtreme Scale can run on different versions of Java 2 Platform, Standard Edition (J2SE). ObjectGrid Version 6.1 supports J2SE Version 1.4.2 and later. For improved developer productivity and performance, use J2SE 5 or later to take advantage of annotations and improved garbage collection. ObjectGrid works on 32-bit or 64-bit JVMs.

ObjectGrid Version 6.0.2 clients can attach to an ObjectGrid Version 6.1 data grid. Use ObjectGrid Version 6.1 clients for J2SE Version 1.4.2 or better clients. The only reason to use an ObjectGrid Version 6.0.2 client is for J2SE Version 1.3 support.

WebSphere eXtreme Scale is tested with a subset of the available virtual machines, however, the supported list is not exclusive. You can run WebSphere eXtreme Scale on any Version 1.4.2 or above, but if a problem on the JVM is identified, you must contact the JVM vendor for support. If possible, use the JVM from the WebSphere runtime on any platform that WebSphere Application Server supports.

Java Platform, Standard Edition 6 is the best JVM. Java 2 Platform, Standard Edition, v 1.4 performs poorly especially for scenarios where the gencon collector makes a difference. Java Platform Standard Edition 5 performs well, but Java Platform, Standard Edition 6 performs better.

orb.properties tuning

The recommendation is to use the following orb.properties file for production. In our lab, we have used this file on data grids of up to 1500 JVMs. The orb.properties file is in the lib folder of the JRE being used.

```
# IBM JDK properties for ORB
org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB
org.omg.CORBA.ORBSingletonClass=com.ibm.rmi.corba.ORBSingleton

# WS Interceptors
org.omg.PortableInterceptor.ORBInitializerClass.com.ibm.ws.objectgrid.corba.ObjectGridInitializer

# WS ORB & Plugins properties
com.ibm.CORBA.ForceTunnel=never
com.ibm.CORBA.RequestTimeout=10
com.ibm.CORBA.ConnectTimeout=10

# Needed when lots of JVMs connect to the catalog at the same time
com.ibm.CORBA.ServerSocketQueueDepth=2048

# Clients and the catalog server can have sockets open to all JVMs
com.ibm.CORBA.MaxOpenConnections=1016

# Thread Pool for handling incoming requests, 200 threads here
com.ibm.CORBA.ThreadPool.IsGrowable=false
com.ibm.CORBA.ThreadPool.MaximumSize=200
com.ibm.CORBA.ThreadPool.MinimumSize=200
com.ibm.CORBA.ThreadPool.InactivityTimeout=180000

# No splitting up large requests/responses in to smaller chunks
com.ibm.CORBA.FragmentSize=0
```

Thread count

The thread count depends on a few factors. A limit exists for how many threads a single shard can manage. With more shards for each JVM, more threads and more concurrency can exist. Each additional shard provides more concurrent paths to the data. Each shard is as concurrent as possible but even so, a limit exists.

CopyMode best practices

WebSphere eXtreme Scale makes a copy of the value based on the six available CopyMode settings. Determine which setting works best for your deployment requirements.

You can use the BackingMap API setCopyMode(CopyMode, valueInterfaceClass) method to set the copy mode to one of the following final static fields that are defined in the com.ibm.websphere.objectgrid.CopyMode class.

When an application uses the ObjectMap interface to obtain a reference to a map entry, use that reference only within the WebSphere eXtreme Scale transaction that obtained the reference. Using the reference in a different transaction can lead to errors. For example, if you use the pessimistic locking strategy for the BackingMap, a get or getForUpdate method call acquires an S (shared) or U (update) lock, depending on the transaction. The get method returns the reference to the value and the lock that is obtained is released when the transaction completes. The transaction must call the get or getForUpdate method to lock the map entry in a different transaction. Each transaction must obtain its own reference to the value by calling the get or getForUpdate method instead of reusing the same value reference in multiple transactions.

CopyMode for entity maps

When using a map associated with an EntityManager API entity, the map always returns the entity Tuple objects directly without making a copy unless you are using COPY_TO_BYTES copy mode. It is important that the CopyMode is updated or the Tuple is copied appropriately when making changes.

COPY_ON_READ_AND_COMMIT

The COPY_ON_READ_AND_COMMIT mode is the default mode. The valueInterfaceClass argument is ignored when this mode is used. This mode ensures that an application does not contain a reference to the value object that is in the BackingMap. Instead, the application is always working with a copy of the value that is in the BackingMap. The COPY_ON_READ_AND_COMMIT mode ensures that the application can never inadvertently corrupt the data that is cached in the BackingMap. When an application transaction calls an ObjectMap.get method for a given key, and it is the first access of the ObjectMap entry for that key, a copy of the value is returned. When the transaction is committed, any changes that are committed by the application are copied to the BackingMap to ensure that the application does not have a reference to the committed value in the BackingMap.

COPY_ON_READ

The COPY_ON_READ mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when a transaction is committed. The valueInterfaceClass argument is ignored when this mode is used. To preserve the integrity of the BackingMap data, the application ensures that every reference that it has for an entry is destroyed after the transaction is committed. With this mode, the ObjectMap.get method returns a copy of the value instead of a reference to the value to ensure that changes that are made by the application to the value does not affect the BackingMap value until the transaction is committed. However, when the transaction does commit, a copy of changes is not made. Instead, the reference to the copy that was returned by the ObjectMap.get method is stored in the BackingMap. The application destroys all map entry references after the transaction is committed. If application does not destroy the map entry references, the application might cause the data cached in BackingMap to become corrupted. If an application is using this mode and is having problems, switch to COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is failing to destroy all of its references after the transaction has committed.

COPY_ON_WRITE

The COPY_ON_WRITE mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when the ObjectMap.get method is called for the first time by a transaction for a given key. The ObjectMap.get method returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface that is specified by the valueInterfaceClass argument. The proxy provides a copy on write implementation. When a transaction commits, the BackingMap examines the proxy to determine if any copy was made as a result of a set method being called. If a copy was made, then the reference to that copy is stored in the BackingMap. The big advantage of this mode is that a value is never copied on a read or at a commit when the transaction never calls a set method to change the value.

The COPY_ON_READ_AND_COMMIT and COPY_ON_READ modes both make a deep copy when a value is retrieved from the ObjectMap. If an application only updates some of the values that are retrieved in a transaction then this mode is not optimal. The COPY_ON_WRITE mode supports this behavior in an efficient manner but requires that the application uses a simple pattern. The value objects are required to support an interface. The application must use the methods on this interface when it is interacting with the value in an eXtreme Scale Session. If this is the case, then eXtreme Scale creates proxies for the values that are returned to the application. The proxy has a reference to be real value. If the application performs read operations only, the read operations always run against the real copy. If the application modifies an attribute on the object, the proxy makes a copy of the real object and then makes the modification on the copy. The proxy then uses the copy from that point on. Using the copy allows the copy operation to be avoided completely for objects that are only read by the application. All modify operations must start with the set prefix. Enterprise JavaBeans normally are coded to use this style of method naming for methods that modify the objects attributes. This convention must be followed. Any objects that are modified are copied at the time that they are modified by the application. This read and write scenario is the most efficient scenario supported by eXtreme Scale. To configure a map to use COPY_ON_WRITE mode, use the following example. In this example, the application stores Person objects that are keyed using the name in the Map. The person object is represented in the following code snippet.

```
class Person {
    String name;
    int age;
    public Person() {
    }
    public void setName(String n)  {
        name = n;
    }
    public String getName() {
        return name;
    }
    public void setAge(int a) {
        age = a;
    }
    public int getAge() {
        return age;
    }
}
```

The application uses the IPerson interface only when it interacts with values that are retrieved from a ObjectMap. Modify the object to use an interface as in the following example.

```
interface IPerson
{
    void setName(String n);
    String getName();
    void setAge(int a);
    int getAge();
}
// Modify Person to implement IPerson interface
class Person implements IPerson {
    ...
}
```

The application then needs to configure the BackingMap to use COPY_ON_WRITE mode, like in the following example:

```
ObjectGrid dg = ...;
BackingMap bm = dg.defineMap("PERSON");
// use COPY_ON_WRITE for this Map with
// IPerson as the valueProxyInfo Class
bm.setCopyMode(CopyMode.COPY_ON_WRITE,IPerson.class);
// The application should then use the following
// pattern when using the PERSON Map.
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
...
sess.begin();
// the application casts the returned value to IPerson and not Person
IPerson p = (IPerson)person.get("Billy");
p.setAge(p.getAge()+1);
...
// make a new Person and add to Map
Person p1 = new Person();
p1.setName("Bobby");
p1.setAge(12);
person.insert(p1.getName(), p1);
sess.commit();
// the following snippet WON'T WORK. Will result in ClassCastException
sess.begin();
// the mistake here is that Person is used rather than
// IPerson
Person a = (Person)person.get("Bobby");
sess.commit();
```

The first section shows the application retrieving a value that was named Billy in the map. The application casts the returned value to the IPerson object, not the Person object because the proxy that is returned implements two interfaces:

- The interface specified in the BackingMap.setCopyMode method call
- The com.ibm.websphere.objectgrid.ValueProxyInfo interface

You can cast the proxy to two types. The last part of the preceding code snippet demonstrates what is not allowed in COPY_ON_WRITE mode. The application retrieves the Bobby record and tries to cast the record to a Person object. This action fails with a class cast exception because the proxy that is returned is not a Person object. The returned proxy implements the IPerson object and ValueProxyInfo.

ValueProxyInfo interface and partial update support: This interface allows an application to retrieve either the committed read-only value referenced by the proxy or the set of attributes that have been modified during this transaction.

```
public interface ValueProxyInfo {
    List /**/ ibmGetDirtyAttributes();
    Object ibmGetRealValue();
}
```

The ibmGetRealValue method returns a read only copy of the object. The application must not modify this value. The ibmGetDirtyAttributes method returns a list of strings representing the attributes that have been modified by the application during this transaction. The main use case for ibmGetDirtyAttributes is in a Java database connectivity (JDBC) or CMP based loader. Only the attributes that are named in the list need be updated on either the SQL statement or object mapped to the table, which leads to more efficient SQL generated by the Loader. When a copy on write transaction is committed and if a loader is plugged in, the the loader can cast the values of the modified objects to the ValueProxyInfo interface to obtain this information.

Handling the equals method when using COPY_ON_WRITE or proxies: For example, the following code constructs a Person object and then inserts it to a an ObjectMap. Next, it retrieves the same object using the ObjectMap.get method. The value is cast to the interface. If the value is cast to the Person interface, a ClassCastException exception results because the returned value is a proxy that implements the IPerson interface and is not a Person object. The equality check fails when using the == operation because they are not the same object.

```
session.begin();
// new the Person object
Person p = new Person(...);
personMap.insert(p.getName, p);
// retrieve it again, remember to use the interface for the cast
IPerson p2 = personMap.get(p.getName());
if(p2 == p) {
    // they are the same
} else {
    // they are not
}
```

Another consideration is when you must override the equals method. As illustrated in the following snippet of code, the equals method must verify that the argument is an object that implements IPerson interface and cast the argument to be a IPerson. Because the argument might be a proxy that implements the IPerson interface, you must use the getAge and getName methods when comparing instance variables for equality.

```
{
    if ( obj == null ) return false;
    if ( obj instanceof IPerson ) {
        IPerson x = (IPerson) obj;
        return ( age.equals( x.getAge() ) && name.equals( x.getName() ) )
    }
    return false;
}
```

ObjectQuery and HashIndex configuration requirements: When using COPY_ON_WRITE with ObjectQuery or a HashIndex plug-in, it's important to configure the ObjectQuery schema and HashIndex plug-in to access the objects using property methods, which is the default. If configured to use field access, the query engine and index will attempt to access the fields in the proxy object, which will always return null or 0 since the object instance will be a proxy.

NO_COPY

The NO_COPY mode allows an application to ensure that it never modifies a value object that is obtained using an ObjectMap.get method in exchange for performance improvements. The valueInterfaceClass argument is ignored when this mode is used. If this mode is used, no copy of the value is ever made. If the application modifies values, then the data in the BackingMap is corrupted. The NO_COPY mode is primarily useful for read-only maps where data is never modified by the application. If the application is using this mode and it is having problems, then switch to the COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is modifying the value returned by ObjectMap.get method, either during transaction or after transaction has committed. All maps associated with EntityManager API entities automatically use this mode regardless of what is specified in the eXtreme Scale configuration.

All maps associated with EntityManager API entities automatically use this mode regardless of what is specified in the eXtreme Scale configuration.

COPY_TO_BYTES

You can store objects in a serialized format instead of POJO format. By using the COPY_TO_BYTES setting, you can reduce the memory footprint that a large graph of Objects can consume. See “Byte array maps” on page 260 for additional information.

Incorrect use of CopyMode

Errors occur when an application attempts to improve performance by using the COPY_ON_READ, COPY_ON_WRITE, or NO_COPY copy mode, as described above. The intermittent errors do not occur when you change the copy mode to the COPY_ON_READ_AND_COMMIT mode.

Problem

The problem might be due to corrupted data in the ObjectGrid map, which is a result of the application violating the programming contract of the copy mode that is being used. Data corruption can cause unpredictable errors to occur intermittently or in an unexplained or unexpected fashion.

Solution

The application must comply with the programming contract that is stated for the copy mode being used. For the COPY_ON_READ and COPY_ON_WRITE copy modes, the application uses a reference to a value object outside of the transaction scope from which the value reference was obtained. To use these modes, the application must delete the reference to the value object after the transaction completes, and obtain a new reference to the value object in each transaction that accesses the value object. For the NO_COPY copy mode, the application must never change the value object. In this case, either write the application so that it does not change the value object, or set the application to use a different copy mode.

Byte array maps

You can store the key-value pairs in your maps in a byte array instead of POJO form, which reduces the memory footprint that a large graph of objects can consume.

Advantages

The amount of memory that is consumed increases with the number of objects in a graph of objects. By reducing a complicated graph of objects to a byte array, only one object is maintained in the heap instead of several objects. With this reduction of the number of objects in the heap, the Java run time has fewer objects to search for during garbage collection.

The default copy mechanism used by WebSphere eXtreme Scale is serialization, which is expensive. For instance, if using the default copy mode of `COPY_ON_READ_AND_COMMIT`, a copy is made both at read time and at get time. Instead of making a copy at read time, with byte arrays, the value is inflated from bytes, and instead of making a copy at commit time, the value is serialized to bytes. Using byte arrays results in equivalent data consistency to the default setting with a reduction of memory used.

When using byte arrays, note that having an optimized serialization mechanism is critical to seeing a reduction of memory consumption. For more information, see "Serialization performance" on page 268.

Configuring byte array maps

You can enable byte array maps with the ObjectGrid XML file by modifying the CopyMode attribute that is used by a map to the setting COPY_TO_BYTES, shown in the following example:

```
<backingMap name="byteMap" copyMode="COPY_TO_BYTES" />
```

See the topic on the ObjectGrid descriptor XML file in the *Administration Guide* for more information.

Considerations

You must consider whether or not to use byte array maps in a given scenario. Although you can reduce your memory use, processor use can increase when you use byte arrays.

The following list outlines several factors that should be considered before choosing to use the byte array map function.

Object type

Comparatively, memory reduction may not be possible when using byte array maps for some object types. Consequently, several types of objects exist for which you should not use byte array maps. If you are using any of the Java primitive wrappers as values, or a POJO that does not contain references to other objects (only storing primitive fields), the number of Java Objects is already as low as possible–there is only one. Since the amount of memory used by the object is already optimized, using a byte array map for these types of objects is not recommended. Byte array maps are more suitable to object types that contain other objects or collections of objects where the total number of POJO objects is greater than one.

For example, if you have a Customer object that had a business Address and a home Address, as well as a collection of Orders, the number of objects in the heap and the number of bytes used by those objects can be reduced by using byte array maps.

Local access

When using other copy modes, applications can be optimized when copies are made if objects are Cloneable with the default ObjectTransformer or when a custom ObjectTransformer is provided with an optimized copyValue method. Compared to the other copy modes, copying on reads, writes, or commit operations will have additional cost when accessing objects locally. For example, if you have a near cache in a distributed topology or are directly accessing a local or server ObjectGrid instance, the access and commit time will increase when using byte array maps due to the cost of serialization. You will see a similar cost in a distributed topology if you use data grid agents or you access the server primary when using the ObjectGridEventGroup.ShardEvents plug-in.

Plug-in interactions

With byte array maps, objects are not inflated when communicating from a client to a server unless the server needs the POJO form. Plug-ins that interact with the map value will experience a reduction in performance due to the requirement to inflate the value.

Any plug-in that uses LogElement.getCacheEntry or LogElement.getCurrentValue will see this additional cost. If you want to get the key, you can use LogElement.getKey, which avoids the additional overhead associated with the LogElement.getCacheEntry().getKey method. The following sections discuss plug-ins in light of the usage of byte arrays.

Indexes and queries

When objects are stored in POJO format, the cost of doing indexing and querying is minimal because the object does not need to be inflated. When using a byte array map you will have the additional cost of inflating the object. In general if your application uses indexes or queries, it is not recommended to use byte array maps unless you only run queries on key attributes.

Optimistic locking

When using the optimistic locking strategy, you will have the additional cost during updates and invalidate operations. This comes from having to inflate the value on the server to get the version value to do optimistic collision checking. If you are just using optimistic locking to guarantee fetch operations and do not need optimistic collision checking, you can use the com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback to disable version checking.

Loader

With a Loader, you will also have the cost in the eXtreme Scale run time from inflating and reserializing the value when it is used by the Loader. You can still use byte array maps with Loaders, but consider the cost of making changes to the value in such a scenario. For example, you can use the byte array feature in the context of a read mostly cache. In this case, the benefit of having less objects in the heap and less memory used will outweigh the cost incurred from using byte arrays on insert and update operations.

ObjectGridEventListener

When using the transactionEnd method in the ObjectGridEventListener plug-in, you will have an additional cost on the server side for remote requests when accessing a LogElement's CacheEntry or current value. If the implementation of the method does not access these fields, then you will not have the additional cost.

Plug-in evictor performance best practices

If you use plug-in evictors, they are not active until you create them and associate them with a backing map. The following best practices will increase performance for least frequently used (LFU) and least recently used (LRU) evictors.

Least frequently used (LFU) evictor

The concept of a LFU evictor is to remove entries from the map that are used infrequently. The entries of the map are spread over a set amount of binary heaps. As the usage of a particular cache entry grows, it becomes ordered higher in the heap. When the evictor attempts a set of evictions it removes only the cache entries that are located lower than a specific point on the binary heap. As a result, the least frequently used entries are evicted.

Least recently used (LRU) evictor

The LRU Evictor follows the same concepts of the LFU Evictor with a few differences. The main difference is that the LRU uses a first in, first out queue (FIFO) instead of a set of binary heaps. Every time a cache entry is accessed, it moves to the head of the queue. Consequently, the front of the queue contains the most recently used map entries and the end becomes the least recently used map entries. For example, the A cache entry is used 50 times, and the B cache entry is used only once right after the A cache entry. In this situation, the B cache entry is at the front of the queue because it was used most recently, and the A cache entry is at the end of the queue. The LRU evictor evicts the cache entries that are at the tail of the queue, which are the least recently used map entries.

LFU and LRU properties and best practices to improve performance

Number of heaps

When using the LFU evictor, all of the cache entries for a particular map are ordered over the number of heaps that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one binary heap that contains all of the ordering for the map. More heaps also speeds up the time that is required for reordering the heaps because each heap has fewer entries. Set the number of heaps to 10% of the number of entries in your BaseMap.

Number of queues

When using the LRU evictor, all of the cache entries for a particular map are ordered over the number of LRU queues that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one queue that contains all of the ordering for the map. Set the number of queues to 10% of the number of entries in your BaseMap.

MaxSize property

When an LFU or LRU evictor begins evicting entries, it uses the MaxSize evictor property to determine how many binary heaps or LRU queue elements to evict. For example, assume that you set the number of heaps or queues to have about 10 map entries in each map queue. If your MaxSize property is set to 7, the evictor evicts 3 entries from each heap or queue object to bring the size of each heap or queue back down to 7. The evictor only evicts map entries from a heap or queue when that heap or queue has more than the MaxSize property value of elements in it. Set the MaxSize to 70% of your heap or queue size. For this example, the value is set to 7. You can get an approximate size of each heap or queue by dividing the number of BaseMap entries by the number of heaps or queues that are used.

SleepTime property

An evictor does not constantly remove entries from your map. Instead it is idle for a set amount of time, only checking the map every n number of seconds, where n refers to the SleepTime property. This property also positively affects performance: running an eviction sweep too often lowers performance because of the resources that are needed for processing them. However, not using the evictor often can result in a map that has entries that are not needed. A map full of entries that are not needed can negatively affect both the memory requirements and processing resources that are required for your map. Setting the eviction sweep interval to fifteen seconds is a good practice for most maps. If the map is written to frequently and is used at a high transaction rate, consider setting the value to a lower time. If the map is accessed infrequently, you can set the time to a higher value.

Example

The following example defines a map, creates a new LFU evictor, sets the evictor properties, and sets the map to use the evictor:

```
//Use ObjectGridManager to create/get the ObjectGrid. Refer to
// the ObjectGridManger section
ObjectGrid objGrid = ObjectGridManager.create............
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LFUEvictor someEvictor = new LFUEvictor();
someEvictor.setNumberOfHeaps(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Using the LRU evictor is very similar to using an LFU evictor. An example follows:

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LRUEvictor someEvictor = new LRUEvictor();
```

```
someEvictor.setNumberOfLRUQueues(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Notice that only two lines are different from the LFUEvictor example.

Locking performance best practices

Locking strategies and transaction isolation settings affect the performance of your applications.

Retrieve a cached instance

See the information about map entry locking in the *Administration Guide* for more information.

Pessimistic locking strategy

Use the pessimistic locking strategy for read and write map operations where keys often collide. The pessimistic locking strategy has the greatest impact on performance.

Read committed and read uncommitted transaction isolation

When you are using pessimistic locking strategy, set the transaction isolation level using the Session.setTransactionIsolation method. For read committed or read uncommitted isolation, use the Session.TRANSACTION_READ_COMMITTED or Session.TRANSACTION_READ_UNCOMMITTED arguments depending on the isolation. To reset the transaction isolation level to the default pessimistic locking behavior, use the Session.setTransactionIsolation method with the Session.REPEATABLE_READ argument.

Read committed isolation reduces the duration of shared locks, which can improve concurrency and reduce the chance for deadlocks. This isolation level should be used when a transaction does not need assurances that read values remain unchanged for the duration of the transaction.

Use an uncommitted read when the transaction does not need to see the committed data.

Optimistic locking strategy

Optimistic locking is the default configuration. This strategy improves both performance and scalability compared to the pessimistic strategy. Use this strategy when your applications can tolerate some optimistic update failures, while still performing better than the pessimistic strategy. This strategy is excellent for read operations and infrequent update applications.

OptimisticCallback plug-in

The optimistic locking strategy makes a copy of the cache entries and compares them as needed. This operation can be expensive because copying the entry might involve cloning or serialization. To implement the fastest possible performance, implement the custom plug-in for non-entity maps.

See for more information. See the information about the OptimisticCallback plug-in in the *Product Overview* for more information.

Use version fields for entities

When you are using optimistic locking with entities, use the @Version annotation or the equivalent attribute in the Entity metadata descriptor file. The version annotation gives the ObjectGrid a very efficient way of tracking the version of an object. If the entity does not have a version field and optimistic locking is used for the entity, then the entire entity must be copied and compared.

None locking strategy

Use the none locking strategy for applications that are read only. The none locking strategy does not obtain any locks or use a lock manager. Therefore, this strategy offers the most concurrency, performance and scalability.

Map entry locks with query and indexes

This topic describes how eXtreme Scale Query APIs and the MapRangeIndex indexing plug-in interact with locks and some best practices to increase concurrency and decrease deadlocks when using the pessimistic locking strategy for maps.

Overview

The ObjectGrid Query API allows SELECT queries over ObjectMap cache objects and entities. When a query is run, the query engine uses a MapRangeIndex when possible to find matching keys that match values in the query's WHERE clause or to bridge relationships. When an index isn't available, the query engine will scan each entry in one or more maps to find the appropriate entries. Both the query engine and index plugins will acquire locks to verify consistent data, depending on the locking strategy, transaction isolation level, and transaction state.

Locking with the HashIndex plug-in

The eXtreme Scale HashIndex plug-in allows finding keys based on a single attribute stored in the cache entry value. The index stores the indexed value in a separate data structure from the cache map. The index validates the keys against map entries before returning to the user to try to achieve an accurate result set. When the pessimistic lock strategy is used and the index is used against a local ObjectMap instance (versus a client/server ObjectMap), the index will acquire locks for each matching entry. When using optimistic locking or a remote ObjectMap, the locks are always immediately released.

The type of lock that is acquired depends upon the forUpdate argument passed to the ObjectMap.getIndex method. The forUpdate argument specifies the type of lock that the index should acquire. If false, a shareable (S) lock is acquired and if true, an upgradeable (U) lock is acquired.

If the lock type is shareable, the transaction isolation setting for the session is applied and affects the duration of the lock. See the transaction isolation topic for details on how transaction isolation is used to add concurrency to applications.

Shared locks with query

The eXtreme Scale query engine acquires S locks when needed to introspect the cache entries to discover if they satisfy the query's filter criteria. When using repeatable read transaction isolation with pessimistic locking, the S locks are only retained for the elements that are included in the query result and are released for any entries that are not included in the result. If using a lower transaction isolation level or optimistic locking, the S locks are not retained.

Shared locks with client to server query

When using the eXtreme Scale query from a client, the query typically runs on the server unless all of the maps or entities referenced in the query are local to the client (for example: a client-replicated map or a query result entity). All queries that run in a read/write transaction will retain S locks as described in the previous section. If the transaction is not a read/write transaction, then a session is not retained on the server and the S locks are released.

A read/write transaction is only routed to a primary partition and a session is maintained on the server for the client session. A transaction can be promoted to read/write under the following conditions:

1. Any map configured to use pessimistic locking is accessed using the ObjectMap get and getAll API methods or the EntityManager.find methods.
2. The transaction is flushed, causing updates to be sent to the server.
3. Any map configured to use optimistic locking is accessed using the ObjectMap.getForUpdate or EntityManager.findForUpdate method.

Upgradeable locks with query

Shareable locks are useful when concurrency and consistency is important. It guarantees that an entry's value does not change for the life of the transaction. No other transaction can change the value while any other S locks are held, and only one other transaction can establish an intent to update the entry. See the Pessimistic Locking Mode topic for details on the S, U and X locking modes.

Upgradeable locks are used to identify the intent to update a cache entry when using the pessimistic lock strategy. It allows synchronization between transactions that want to modify a cache entry. Transactions can still view the entry using an S lock, but other transactions are prevented from acquiring a U lock or an X lock. In many scenarios, acquiring a U lock without first acquiring an S lock is necessary to avoid deadlocks. See the Pessimistic Locking Mode topic for common deadlock examples.

The ObjectQuery and EntityManager Query interfaces provide the setForUpdate method to identify the intended use for the query result. Specifically, the query engine acquires U locks instead of S locks for each map entry involved in the query result:

```
ObjectMap orderMap = session.getMap("Order");
ObjectQuery q = session.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
session.begin();
// Run the query.  Each order has  U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
    orderMap.update(o.getId(), o);
}
// When committed, the
session.commit();
```

```
Query q = em.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
emTran.begin();
// Run the query.  Each order has  U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
}
tmTran.commit();
```

When the **setForUpdate** attribute is enabled, the transaction is automatically converted to a read/write transaction and the locks are held on the server as expected. If the query cannot use any indexes, then the map must be scanned which will result in temporary U locks for map entries that do not satisfy the query result, and hold U locks for entries that are included in the result.

ObjectTransformer interface best practices

The ObjectTransformer interface uses callbacks to the application to provide custom implementations of common and expensive operations such as object serialization and deep copies on objects.

Overview

For details about the ObjectTransformer interface, see “ObjectTransformer plug-in” on page 197. From a performance viewpoint, and from the CopyMode method information that is in the CopyMode method best practices topic, eXtreme Scale clearly copies the values for all cases except when NO_COPY mode is used. The default copying mechanism that is employed in eXtreme Scale is serialization, which is known as an expensive operation. The ObjectTransformer interface is used in this situation. The ObjectTransformer interface uses callbacks to the application to provide a custom implementation of common and expensive operations, such as object serialization and deep copies on objects.

An application can provide an implementation of the ObjectTransformer interface to a map, and eXtreme Scale then delegates to the methods on this object and relies on the application to provide an optimized version of each method in the interface. The ObjectTransformer interface follows:

```
public interface ObjectTransformer {
    void serializeKey(Object key, ObjectOutputStream stream) throws IOException;
    void serializeValue(Object value, ObjectOutputStream stream) throws IOException;
    Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object copyValue(Object value);
    Object copyKey(Object key);
}
```

You can associate an ObjectTransformer interface with a BackingMap by using the following example code:

```
ObjectGrid g = ...;
BackingMap bm = g.defineMap("PERSON");
MyObjectTransformer ot = new MyObjectTransformer();
bm.setObjectTransformer(ot);
```

Tune object serialization and inflation

Object serialization is typically the most important performance consideration with eXtreme Scale, which uses the default serializable mechanism if an ObjectTransformer plug-in is not supplied by the application. An application can provide implementations of either the Serializable readObject and writeObject, or it can have the objects implement the Externalizable interface, which is approximately ten times faster. If the objects in the map cannot be modified, then an application can associate an ObjectTransformer interface with the ObjectMap. The serialize and inflate methods are provided to allow the application to provide custom code to optimize these operations, given their large performance impact on the system. The serialize method serializes the object to the provided stream. The inflate method provides the input stream and expects the application to create the object, inflate it using data in the stream and return the object. Implementations of the serialize and inflate methods must mirror each other.

Tune deep copy operations

After an application receives an object from an ObjectMap, eXtreme Scale performs a deep copy on the object value to ensure that the copy in the BaseMap map maintains data integrity. The application can then modify the object value safely. When the transaction commits, the copy of the object value in the BaseMap map is updated to the new modified value and the application stops using the value from that point on. You could have copied the object again at the commit phase to make a private copy. However, in this case the performance cost of this action was traded off against requiring the application programmer not to use the value after the transaction commits. The default ObjectTransformer attempts to use either a clone or a serialize and inflate pair to generate a copy. The serialize and inflate pair is the worst case performance scenario. If profiling reveals that serialize and inflate is a problem for your application, write an appropriate clone method to create a deep copy. If you cannot alter the class, then create a custom ObjectTransformer plug-in and implement more efficient copyValue and copyKey methods.

Serialization performance

WebSphere eXtreme Scale uses multiple Java processes to hold data. These processes serialize the data: That is, they convert the data (which is in the form of Java object instances) to bytes and back to objects again as needed to move the data between client and server processes. Marshalling the data is the most expensive operation and must be addressed by the application developer when designing the schema, configuring the data grid and interacting with the data-access APIs.

The default Java serialization and copy routines are relatively slow and can consume 60 to 70 percent of the processor in a typical setup. The following sections are choices for improving the performance of the serialization.

Write an ObjectTransformer for each BackingMap

An ObjectTransformer can be associated with a BackingMap. Your application can have a class that implements the ObjectTransformer interface and provides implementations for the following operations:

- Copying values
- Serializing and inflating keys to and from streams
- Serializing and inflating values to and from streams

The application does not need to copy keys because keys are considered immutable.

For more information, see “ObjectTransformer plug-in” on page 197 and “ObjectTransformer interface best practices” on page 267.

Note: The ObjectTransformer is only invoked when the ObjectGrid knows about the data that is being transformed. For example, when DataGrid API agents are used, the agents themselves as well as the agent instance data or data returned from the agent must be optimized using custom serialization techniques. The ObjectTransformer is not invoked for DataGrid API agents.

Using entities

When using the EntityManager API with entities, the ObjectGrid does not store the entity objects directly into the BackingMaps. The EntityManager API converts the entity object to Tuple objects. See For more information, see the topic on using a loader with entity maps and tuples in the *Programming Guide*. Entity maps are automatically associated with a highly optimized ObjectTransformer. Whenever the ObjectMap API or EntityManager API is used to interact with entity maps, the entity ObjectTransformer is invoked.

Custom serialization

There are some cases when objects must be modified to use custom serialization, such as implementing the java.io.Externalizable interface or by implementing the writeObject and readObject methods for classes implementing the java.io.Serializable interface. Custom serialization techniques should be employed when the objects are serialized using mechanisms other than the ObjectGrid API or EntityManager API methods.

For example, when objects or entities are stored as instance data in a DataGrid API agent or the agent returns objects or entities, those objects are not transformed using an ObjectTransformer. The agent, will however, automatically use the ObjectTransformer when using `EntityMixin`interface. See DataGrid agents and entity based Maps for further details.

Byte arrays

When using the ObjectMap or DataGrid APIs, the key and value objects are serialized whenever the client interacts with the data grid and when the objects are replicated. To avoid the overhead of serialization, use byte arrays instead of Java objects. Byte arrays are much cheaper to store in memory since the JDK has less objects to search for during garbage collection and they are can be inflated only when needed. Byte arrays should only be used if you do not need to access the objects using queries or indexes. Since the data is stored as bytes, the data can only be accessed through its key.

WebSphere eXtreme Scale can automatically store data as byte arrays using the CopyMode.COPY_TO_BYTES map configuration option, or it can be handled manually by the client. This option will store the data efficiently in memory and can also automatically inflate the objects within the byte array for use by query and indexes on demand.

Chapter 9. Troubleshooting

To troubleshoot the configuration of your eXtreme Scale in-memory data grid, you can use logs and trace, messages, and release notes.

Related concepts

Troubleshooting XML configuration

Logs and trace

You can use logs and trace to monitor and troubleshoot your environment. Logs are in different locations depending on your configuration. You might need to provide trace for a server when you work with IBM support.

Logs with WebSphere Application Server

See the WebSphere Application Server Information Center for more information.

Logs with WebSphere eXtreme Scale in a stand-alone environment

With stand-alone catalog and container servers, you set the location of logs and any trace specification. The catalog server logs are in the location where you ran the start server command.

Setting the log location for container servers

By default, the logs for a container are in the directory where the server command was run. If you start the servers in the `<`*`eXtremeScale_home`*`>/bin` directory, the logs and trace files are in the `logs/<`*`server_name`*`>` directories in the `bin` directory. To specify an alternate location of a container server logs, create a properties file, such as a `server.properties` file, with the following contents:

```
workingDirectory=<directory>
traceSpec=
systemStreamToFileEnabled=true
```

The workingDirectory property is the root directory for the logs and optional trace file. WebSphere eXtreme Scale creates a directory with the name of the container server with a `SystemOut.log` file, a `SystemErr.log` file, and a trace file if trace was enabled with the traceSpec option. To use a properties file during container startup, use the **`-serverProps`** option and provide the server properties file location.

Common information messages to look for in the `SystemOut.log` file are start confirmation messages. For more information about a specific message, see "Messages" on page 277.

Trace with WebSphere Application Server

See the WebSphere Application Server Information Center for more information.

Trace on a stand-alone catalog service

You can set trace on a catalog service by using the **-traceSpec** and **-traceFile** parameters during catalog service startup. For example:

```
startOgServer.sh catalogServer -traceSpec
ObjectGridPlacement=all=enabled -traceFile
/home/user1/logs/trace.log
```

If you start the catalog service in the *<eXtremeScale_home>*/bin directory, the logs and trace files will be in a logs/*<catalog_service_name>* directory in the bin directory. See the information about starting the catalog service process in a stand-alone environment in the *Administration Guide*.

Trace on a stand-alone container server

You can enable trace on a container server in two ways. You can create a server properties file as explained in the logs section, or you can enable trace by using the command line on startup. To enable container trace with a server properties file, update the **traceSpec** property with the required trace specification. To enable container trace using start parameters, use the **-traceSpec**and **-traceFile** parameters. For example:

```
startOgServer.sh c0 -objectGridFile ../xml/myObjectGrid.xml
-deploymentPolicyFile ../xml/myDepPolicy.xml -catalogServiceEndpoints
server1.rchland.ibm.com:2809 -traceSpec
ObjectGridPlacement=all=enabled -traceFile /home/user1/logs/trace.log
```

If you start the server in the *<eXtremeScale_home>*/bin directory, the logs and trace files are in the logs/*<server_name>* directories in the bin directory

See

Trace on a stand-alone client

You can start trace collection on a stand-alone client by adding system properties to the startup script for the client application. In the following example, trace settings are specified for the com.ibm.samples.MyClientProgram application:

```
java -DtraceSettingsFile=MyTraceSettings.properties
-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager
-Djava.util.logging.configureByServer=true com.ibm.samples.MyClientProgram
```

See WebSphere Application Server: Enabling trace on client and stand-alone applications for more information.

Trace with the ObjectGridManager interface

Another option is to set trace during run time on an ObjectGridManager interface. Setting trace on an ObjectGridManager interface can be used to get trace on an eXtreme Scale client while it connects to an eXtreme Scale and commits transactions. To set trace on an ObjectGridManager interface, supply a trace specification and a trace log.

```
ObjectGridManager manager = ObjectGridManagerFactory.getObjectGridManager();
...
manager.setTraceEnabled(true);
manager.setTraceFileName("logs/myClient.log");
manager.setTraceSpecification("ObjectGridReplication=all=enabled");
```

Trace with the xsadmin utility

To enable trace with the xsadmin utility, use the **`setTraceSpec`** option. Use the xsadmin utility to enable trace on a stand-alone environment during run time instead of during startup. You can enable trace with xsadmin on container servers only:

```
xsadmin.bat -setTraceSpec "ObjectGridReplication=all=enabled"
```

You can also disable trace by setting the trace specification to `*=all=disabled.`

See the information about the xsAdmin utility in the *Administration Guide* for more information.

ffdc directory and files

FFDC files are for IBM support to aid in debug. These files might be requested by IBM support if a problem occurs.

These files are in a directory labeled, `ffdc`, and contain files that resemble the following:

```
server2_exception.log
server2_20802080_07.03.05_10.52.18_0.txt
```

Trace options

You can enable trace to provide information about your environment to IBM support.

About trace

WebSphere eXtreme Scale trace is divided into several different components. Similarly toWebSphere Application Server trace, you can specify the level of trace to use. Common levels of trace include: all, debug, entryExit, and event.

An example trace string follows:

```
ObjectGridComponent=level=enabled
```

You can concatenate trace strings. Use the `*` (asterisk) symbol to specify a wildcard value, such as `ObjectGrid*=all=enabled`. If you need to provide a trace to IBM support, a specific trace string is requested. For example, if a problem with replication occurs, the `ObjectGridReplication=debug=enabled` trace string might be requested.

Trace specification

ObjectGrid
General core cache engine.

ObjectGridCatalogServer
General catalog service.

ObjectGridChannel
Static deployment topology communications.

ObjectgridCORBA
Dynamic deployment topology communications.

ObjectGridDataGrid
The AgentManager API.

ObjectGridDynaCache
The WebSphere eXtreme Scale dynamic cache provider.

ObjectGridEntityManager
The EntityManager API. Use with the Projector option.

ObjectGridEvictors
ObjectGrid built-in evictors.

ObjectGridJPA
Java Persistence API (JPA) loaders.

ObjectGridJPACache
JPA cache plug-ins.

ObjectGridLocking
ObjectGrid cache entry lock manager.

ObjectGridMBean
Management beans.

ObjectGridPlacement
Catalog server shard placement service.

ObjectGridQuery
ObjectGrid query.

ObjectGridReplication
Replication service.

ObjectGridRouting
Client/server routing details.

ObjectGridSecurity
Security trace.

ObjectGridStats
ObjectGrid statistics.

ObjectGridStreamQuery
The Stream Query API.

ObjectGridWriteBehind
ObjectGrid write behind.

Projector
The engine within the EntityManager API.

QueryEngine
The query engine for the Object Query API and EntityManager Query API.

QueryEnginePlan
Query plan diagnostics.

Troubleshooting loaders

Use this information to troubleshoot issues with your database loaders.

Procedure

- **Problem:** When you are using an OpenJPA loader with DB2 in WebSphere Application Server, a closed cursor exception occurs.

  The following exception is from DB2 in the org.apache.openjpa.persistence.PersistenceException log file:

  ```
  [jcc][t4][10120][10898][3.57.82] Invalid operation: result set is closed.
  ```

Solution: By default, the application server configures the resultSetHoldability custom property with a value of `2` (CLOSE_CURSORS_AT_COMMIT). This property causes DB2 to close its resultSet/cursor at transaction boundaries. To remove the exception, change the value of the custom property to `1` (HOLD_CURSORS_OVER_COMMIT). Set the resultSetHoldability custom property on the following path in the WebSphere Application Server cell: **Resources** > **JDBC provider** > **DB2 Universal JDBC Driver Provider** > **DataSources** > *data_source_name* > **Custom properties** > **New**.

- **Problem:** DB2 displays an exception: `The current transaction has been rolled back because of a deadlock or timeout. Reason code "2".. SQLCODE=-911, SQLSTATE=40001, DRIVER=3.50.152`

  This exception occurs because of a lock contention problem when you are running with OpenJPA with DB2 in WebSphere Application Server. The default isolation level for WebSphere Application Server is Repeatable Read (RR), which obtains long-lived locks with DB2.

  Solution: Set the isolation level to Read Committed to reduce the lock contention. Set the webSphereDefaultIsolationLevel data source custom property to set the isolation level to `2`(TRANSACTION_READ_COMMITTED) on the following path in the WebSphere Application Server cell: **Resources** > **JDBC provider** > *JDBC_provider* > **Data sources** > *data_source_name* > **Custom properties** > **New**. For more information about the webSphereDefaultIsolationLevel custom property and transaction isolation levels, see Requirements for setting data access isolation levels.
- **Problem:** When you are using the preload function of the JPALoader or JPAEntityLoader, the following CWOBJ1511 message does not display for the partition in a container server: `CWOBJ1511I: GRID_NAME:MAPSET_NAME:PARTITION_ID (primary) is open for business`.

  Instead, a TargetNotAvailableException exception occurs in the container server, which activates the partition that is specified by the preloadPartition property.

  Solution: Set the preloadMode attribute to `true` if you use a JPALoader or JPAEntityLoader to preload data into the map. If the preloadPartition property of the JPALoader and JPAEntityLoader is set to a value between `0` and `total_number_of_partitions - 1`, then the JPALoader and JPAEntityLoader try to preload the data from backend database into the map. The following snippet of code illustrates how the preloadMode attribute is set to enable asynchronous preload:

  ```
  BackingMap bm = og.defineMap( "map1" );
  bm.setPreloadMode( true );
  ```

  You can also set the preloadMode attribute by using an XML file as illustrated in the following example:

  ```
   <backingMap name="map1" preloadMode="true" pluginCollectionRef="map1"
    lockStrategy="OPTIMISTIC" />
  ```

Related concepts

"Writing a loader" on page 159
You can write your own loader plug-in implementation in your applications, which must follow the common WebSphere eXtreme Scale plug-in conventions.

"Using a Loader" on page 157
With an eXtreme Scale Loader plug-in, an ObjectGrid map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or some other system. Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data, allowing hub-based caches to be built using ObjectGrid. A loader has the logic for reading and writing data to and from a persistent store.

"Client-based JPA preload utility overview" on page 180
The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid.

Loaders overview

Related reference

"Client-based JPA preload utility programming" on page 182
The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid. You can implement preloading and reloading of data in your application.

"JPA loader programming considerations" on page 164
A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

Troubleshooting client connectivity problems

There are several common problems specific to clients and client connectivity that you can solve as described in the following sections.

Procedure

Problem: If you are using the EntityManager API or byte array maps with the COPY_TO_BYTES copy mode, client data access methods result in various serialization-related exceptions or a NullPointerException.

- The following error occurs when you are using the COPY_TO_BYTES copy mode:

```
java.lang.NullPointerException
        at com.ibm.ws.objectgrid.map.BaseMap$BaseMapObjectTransformer2.inflateObject(BaseMap.java:5278)
        at com.ibm.ws.objectgrid.map.BaseMap$BaseMapObjectTransformer.inflateValue(BaseMap.java:5155)
```

- The following error occurs when you are using the EntityManager API:

```
java.lang.NullPointerException
    at com.ibm.ws.objectgrid.em.GraphTraversalHelper.fluffFetchMD(GraphTraversalHelper.java:323)
    at com.ibm.ws.objectgrid.em.GraphTraversalHelper.fluffFetchMD(GraphTraversalHelper.java:343)
      at com.ibm.ws.objectgrid.em.GraphTraversalHelper.getObjectGraph(GraphTraversalHelper.java:102)
       at com.ibm.ws.objectgrid.ServerCoreEventProcessor.getFromMap(ServerCoreEventProcessor.java:709)
      at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processGetRequest(ServerCoreEventProcessor.java:323)
```

Cause: The EntityManager API and COPY_TO_BYTES copy mode use a metadata repository that is embedded in the data grid. When clients connect, the data grid stores the repository identifiers in the client and caches the identifiers for the duration of the client connection. If you restart the data grid, you lose all metadata and the regenerated identifiers do not match the cached identifiers on the client.
Solution: If you are using the EntityManager API or the COPY_TO_BYTES copy mode, disconnect and reconnect all of the clients if the ObjectGrid is stopped and restarted. Disconnecting and reconnecting the clients refreshes the metadata

identifier cache. You can disconnect clients by using the ObjectGridManager.disconnect() method or the ObjectGrid.destroy() method.

Messages

When you encounter a message in a log or other parts of the product interface, you can look up the message by its component prefix to find out more information.

Finding messages

When you encounter a message in a log, copy the message number with its letter prefix and number and search in the information center (for example, `CWOBJ1526I`). When you search for the message, you can find an additional explanation of the message and possible actions you can take to resolve the problem.

See the information center for an index of product messages.

Release notes

Links are provided to the product support Web site, to product documentation, and to last minute updates, limitations, and known problems for the product.

- "Accessing last-minute updates, limitations, and known problems"
- "Accessing system and software requirements"
- "Accessing product documentation"
- "Accessing the product support Web site"
- "Contacting IBM Software Support"

Accessing last-minute updates, limitations, and known problems

The release notes are available on the product support site as technotes. To see a list of all the technotes for WebSphere eXtreme Scale, go to the Support Web page.

- To see a list of the release notes for Version 7.0, go to the Support Web page.
- To see a list of the release notes for Version 6.1, go to the Release notes wiki page.

Accessing system and software requirements

The hardware and software requirements are documented on the following pages:

- Detailed system requirements

Accessing product documentation

For the entire information set, go to the Library page.

Accessing the product support Web site

To search for the latest technotes, downloads, fixes, and other support-related information, go to the Support page.

Contacting IBM Software Support

If you encounter a problem with the product, first try the following actions:

- Follow the steps described in the product documentation

- Look for related documentation in the online help
- Look up error messages in the message reference

If you cannot resolve your problem by any of the preceding methods, contact IBM Technical Support.

Chapter 10. Glossary

This glossary includes terms and definitions for WebSphere eXtreme Scale.

The following cross-references are used in this glossary:
1. See refers the reader from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
2. See also refers the reader to a related or contrasting term.

To view glossaries for other IBM products, go to www.ibm.com/software/globalization/terminology.

administrator. A person responsible for administrative tasks such as access authorization and content management. Administrators can also grant levels of authority to users.

agent. A program that performs an action on behalf of a user or other program without user intervention or on a regular schedule, and reports the results back to the user or program.

APAR. See authorized program analysis report.

API. See application programming interface.

application. One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application programming interface (API). An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

application server. A server program in a distributed network that provides the execution environment for an application program.

asynchronous. Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals.

asynchronous messaging. A method of communication between programs in which a program places a message on a message queue, then proceeds with its own processing without waiting for a reply to its message.

asynchronous replica. A shard that receives updates after the transaction commits. This method is faster than a synchronous replica, but introduces the possibility of data loss because the asynchronous replica can be several transactions behind the primary shard.

authenticated user. A portal user who has logged in to the portal with a valid account (user ID and password). Authenticated users have access to all public places.

authentication. A security service that provides proof that a user of a computer system is genuinely who that person claims to be. Common mechanisms for implementing this service are passwords and digital signatures. Authentication is distinct from authorization; authentication is not concerned with granting or denying access to system resources.

authentication alias. An alias that authorizes access to resource adapters and data sources. An authentication alias contains authentication data, including a user ID and password.

authorization. The process of granting a user, system, or process either complete or restricted access to an object, resource, or function.

authorization policy. A policy whose policy target is a business service and whose contract contains one or more assertions that grant permission to run a channel action.

authorization table. A table that contains the role to user or group mapping information that identifies the permitted access of a client to a particular resource.

authorized program analysis report (APAR). A request for correction of a defect in a supported release of an IBM-supplied program.

autodiscovery. The discovery of service artifacts in a file system, external registry, or another source.

autonomic manager. A set of software or hardware components, configured by policies, which manage the behavior of other software or hardware components as a human might manage them. An autonomic manager includes a control loop that consists of monitor, analyze, plan, and execute components.

availability.

1. The condition allowing users to access and use their applications and data.

2. The time periods during which a resource is accessible. For example, a contractor might have an availability of 9 AM to 5 PM every weekday, and 9 AM to 3 PM on Saturdays.

bean. A definition or instance of a JavaBeans component. See also JavaBeans, enterprise bean.

bean class. In Enterprise JavaBeans (EJB) programming, a Java class that implements a javax.ejb.EntityBean class or javax.ejb.SessionBean class.

Bean Scripting Framework. An architecture for incorporating scripting language functions to Java applications.

bean-managed messaging. A function of asynchronous messaging that gives an enterprise bean complete control over the messaging infrastructure.

bean-managed persistence (BMP). The mechanism whereby data transfer between an entity bean's variables and a resource manager is managed by the entity bean. (Sun)

bean-managed transaction (BMT). The capability of the session bean, servlet, or application client component to manage its own transactions directly, instead of through a container.

binary format. Representation of a decimal value in which each field must be 2 or 4 bytes long. The sign (+ or -) is in the far left bit of the field, and the number value is in the remaining bits of the field. Positive numbers have a 0 in the sign bit and are in true form. Negative numbers have a 1 in the sign bit and are in twos complement form.

BMP. See bean-managed persistence.

BMT. See bean-managed transaction.

bootstrap. A small program that loads larger programs during system initialization.

bootstrapping. The process by which an initial reference of the naming service is obtained. The bootstrap setting and the host name form the initial context for Java Naming and Directory Interface (JNDI) references.

bottleneck. A place in the system where contention for a resource is affecting performance.

bottom-up development. In Web services, the process of developing a service from an existing artifact such as a Java bean or enterprise bean rather than a Web Services Description Language (WSDL) file.

breakpoint. A marked point in a process or programmatic flow that causes that flow to pause when the point is reached, typically to allow debugging or monitoring.

build definition file. An XML file that identifies components and characteristics for a customized installation package (CIP).

build path. The path that is used during compilation of Java source code, in order to find referenced classes that reside in other projects.

build plan. An XML file that defines the processing necessary to build generation outputs and that specifies the machine where processing takes place.

build time data. Objects that are not used by the translator, such as EDI standards, record oriented data document types, and maps.

bytecode. Machine-independent code generated by the Java compiler and executed by the Java interpreter. (Sun)

cache instance resource. A location where any Java Platform, Enterprise Edition (Java EE) application can store, distribute, and share data.

cache replication. The sharing of cache IDs, cache entries, and cache invalidations with other servers in the same replication domain.

catalog. A container that, depending on the container type, holds processes, data, resources, organizations, or reports in the project tree.

catalog service. A service that controls placement of shards and discovers and monitors the health of containers.

category. A container used in a structure diagram to group elements based on a shared attribute or quality.

cell.

1. A group of managed processes that are federated to the same deployment manager and can include high-availability core groups.

2. One or more processes that each host runtime components. Each has one or more named core groups.

cell-scoped binding. A binding scope where the binding is not specific to, and not associated with any node or server. This type of name binding is created under the persistent root context of a cell.

chassis. The metal frame in which various electronic components are mounted.

child node. A node within the scope of another node.

CIP. See customized installation package.

class. In object-oriented design or programming, a model or template that can be used to create objects with a common definition and common properties, operations, and behavior. An object is an instance of a class.

class file. A compiled Java source file.

class hierarchy. The relationships between classes that share a single inheritance.

class loader. Part of the Java virtual machine (JVM) that is responsible for finding and loading class files. A class loader affects the packaging of applications and the runtime behavior of packaged applications deployed on application servers.

class path. A list of directories and JAR files that contain resource files or Java classes that a program can load dynamically at run time.

classifier. A specialized attribute used for grouping and color-coding process elements.

client. A software program or computer that requests services from a server. See also host.

client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

client/server. Pertaining to the model of interaction in distributed data processing in which a program on one computer sends a request to a program on another computer and awaits a response. The requesting program is called a client; the answering program is called a server.

Cloudscape. An embeddable, all Java, object-relational database management system (ORDBMS).

cluster. A group of application servers that collaborate for the purposes of workload balancing and failover.

coarse-grained. Pertaining to viewing a group of objects from an abstract or high level.

coherent cache. Cache that maintains integrity so that all clients see the same data.

collection certificate store. A collection of intermediate certificates or certificate revocation lists (CRL) that are used by a certificate path to build up a certificate chain for validation.

comma delimited file. A file whose records contain fields that are separated by a comma.

command bean. A proxy that can invoke a single operation using an execute() method.

command line. The blank line on a display where commands, option numbers, or selections can be entered.

compilation unit. A portion of a computer program sufficiently complete to be compiled correctly.

compile time. The time period during which a computer program is being compiled into an executable program.

component.

1. A reusable object or program that performs a specific function and works with other components and applications.
2. In Eclipse, one or more plug-ins that work together to deliver a discrete set of functions.

component element. An entity in a component where a breakpoint can be set, such as an activity or Java snippet in a business process, or a mediation primitive or node in a mediation flow.

component instance. A running component that can be running in parallel with other instances of the same component.

component test. An automated test of one or more components of an enterprise application, which may include Java classes, EJB beans, or Web services.

container server. A server instance that can host multiple shards. One Java virtual machine (JVM) can host multiple container servers.

converter. In Enterprise JavaBeans (EJB) programming, a class that translates a database representation to an object type and back.

create method. In enterprise beans, a method defined in the home interface and invoked by a client to create an enterprise bean. (Sun)

credential. In the Java Authentication and Authorization Service (JAAS) framework, a subject class that owns security-related attributes. These attributes can contain information used to authenticate the subject to new services.

customized installation package (CIP). A customized installation image that can include one or more maintenance packages, a configuration archive file from a stand-alone server profile, one or more enterprise archive files, scripts, and other files that help customize the resulting installation.

daemon. A program that runs unattended to perform continuous or periodic functions, such as network control.

dashboard. A Web page that can contain one or more viewers that graphically represent business data.

data grid. A system for accessing terabytes or petrabytes of data.

DB2. A family of IBM licensed programs for relational database management.

deadlock. A condition in which two independent threads of control are blocked, each waiting for the other to take some action. Deadlock often arises from adding synchronization mechanisms to avoid race conditions.

demilitarized zone (DMZ). A configuration that includes multiple firewalls to add layers of protection between a corporate intranet and a public network, such as the Internet.

deploy. To place files or install software into an operational environment. In Java Platform, Enterprise Edition (Java EE), this involves creating a deployment descriptor suitable to the type of application that is being deployed.

deploy phase. See deployment phase.

deployment code. Additional code that enables bean implementation code written by an application developer to work in a particular EJB runtime environment. Deployment code can be generated by tools that the application server vendor supplies.

deployment descriptor. An XML file that describes how to deploy a module or application by specifying configuration and container options. For example, an EJB deployment descriptor passes information to an EJB container about how to manage and control an enterprise bean.

deployment directory. The directory where the published server configuration and Web application are located on the machine where the application server is installed.

deployment environment. A collection of configured clusters, servers, and middleware that collaborate to provide an environment to host software modules. For example, a deployment environment might include a host for message destinations, a processor or sorter of business events, and administrative programs.

deployment manager. A server that manages operations for a logical group or cell of other servers.

deployment phase. A phase that includes a combination of creating the hosting environment for your applications and the deployment of those applications. This includes resolving the application's resource dependencies, operational conditions, capacity requirements, and integrity and access constraints.

deployment policy. An optional way to configure an eXtreme Scale environment based on various items, including: number of systems, servers, partitions, replicas (including type of replica), and heap sizes for each server.

deployment topology. The configuration of servers and clusters in a deployment environment and the physical and logical relationships among them.

deprecated. Pertaining to an entity, such as a programming element or feature, that is supported but no longer recommended and that might become obsolete.

derivation. In object-oriented programming, the refinement or extension of one class from another.

deserialization. A method for converting a serialized variable into object data.

destination. An exit point that is used to deliver documents to a back-end system or a trading partner.

digital certificate. An electronic document used to identify an individual, a system, a server, a company, or some other entity, and to associate a public key with the entity. A digital certificate is issued by a certification authority and is digitally signed by that authority.

dirty read. A read request that does not involve any locking mechanism. This means that data can be read that might later be rolled back resulting in an inconsistency between what was read and what is in the database.

distributed eXtreme Scale. A usage pattern for interacting with eXtreme Scale when servers and clients exist on multiple processes.

DMZ. See demilitarized zone.

DNS. See Domain Name System.

do-while loop. A loop that repeats the same sequence of activities as long as some condition is satisfied. Unlike a while loop, a do-while loop tests its condition at the end of the loop. This means that its sequence of activities always runs at least once.

document type definition (DTD). The rules that specify the structure for a particular class of SGML or XML documents. The DTD defines the structure with elements, attributes, and notations, and it establishes constraints for how each element, attribute, and notation can be used within the particular class of documents.

domain. An object, icon, or container that contains other objects representing the resources of a domain. The domain object can be used to manage those resources.

Domain Name System (DNS). The distributed database system that maps domain names to IP addresses.

downstream. Pertaining to the direction of the flow, which is from the first node in the process (upstream) toward the last node in the process (downstream).

drop-down. See pull-down.

DTD. See document type definition.

DTD document definition. A description or layout of an XML document based on an XML DTD.

dynamic cache. A consolidation of several caching activities, including servlets, Web services, and WebSphere commands into one service where these activities share configuration parameters and work together to improve performance.

dynamic cluster. A server cluster that uses weights to balance the workloads of its cluster members dynamically, based on performance information collected from cluster members.

EAR. See enterprise archive.

EAR project. See enterprise application project.

Eclipse. An open-source initiative that provides ISVs and other tool developers with a standard platform for developing plug-compatible application development tools.

edition. A successive deployment generation of a particular set of versioned artifacts.

editor area. In Eclipse and Eclipse-based products, the area in the workbench window where files are opened for editing.

EJB. See Enterprise JavaBeans.

EJB container. A container that implements the EJB component contract of the Java EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. (Sun)

EJB context. In enterprise beans, an object that allows an enterprise bean to invoke services provided by the container and to obtain information about the caller of a client-invoked method. (Sun)

EJB factory. An access bean that simplifies the creating or finding of an enterprise bean instance.

EJB home object. In Enterprise JavaBeans (EJB) programming, an object that provides the life cycle operations (create, remove, find) for an enterprise bean. (Sun)

EJB inheritance. A form of inheritance in which an enterprise bean inherits properties, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group.

EJB JAR file. A Java archive that contains an EJB module. (Sun)

EJB module. A software unit that consists of one or more enterprise beans and an EJB deployment descriptor. (Sun)

EJB object. In enterprise beans, an object whose class implements the enterprise bean remote interface (Sun).

EJB project. A project that contains the resources needed for EJB applications, including enterprise beans; home, local, and remote interfaces; JSP files; servlets; and deployment descriptors.

EJB query. In EJB query language, a string that contains an optional SELECT clause specifying the EJB objects to return, a FROM clause that names the bean collections, an optional WHERE clause that contains search predicates over the collections, an optional ORDER BY clause that specifies the ordering of the result collection, and input parameters that correspond to the arguments of the finder method.

EJB reference. A logical name used by an application to locate the home interface of an enterprise bean in the target operational environment.

EJB server. Software that provides services to an EJB container. An EJB server may host one or more EJB containers. (Sun)

embedded server. A catalog service or container server that resides in an existing process and is started and stopped within the process.

endpoint.

1. A JCA application or other client consumer of an event from the enterprise information system.

2. The system that is the origin or destination of a session.

endpoint listener. The point or address at which incoming messages for a Web service are received by a service integration bus.

enterprise application project (EAR project). A structure and hierarchy of folders and files that contain a deployment descriptor and IBM extension document as well as files that are common to all Java EE modules that are defined in the deployment descriptor.

enterprise archive (EAR). A specialized type of JAR file, defined by the Java EE standard, used to deploy Java EE applications to Java EE application servers. An EAR file contains EJB components, a deployment descriptor, and Web archive (WAR) files for individual Web applications. See also Web archive.

enterprise bean. A component that implements a business task or business entity and resides in an EJB container. Entity beans, session beans, and message-driven beans are all enterprise beans. (Sun) See also bean.

Enterprise JavaBeans (EJB). A component architecture defined by Sun Microsystems for the development and deployment of object-oriented, distributed, enterprise-level applications (Java EE).

enterprise service bus (ESB). A flexible connectivity infrastructure for integrating applications and services; it offers a flexible and manageable approach to service-oriented architecture implementation.

entity.

1. A simple Java class that represents a row in a database table or entry in a map.

2. In markup languages such as XML, a collection of characters that can be referenced as a unit, for example to incorporate often-repeated text or special characters within a document.

entity bean. In EJB programming, an enterprise bean that represents persistent data maintained in a database. Each entity bean carries its own identity. (Sun)

entry breakpoint. A breakpoint set on a component element that is hit before the component element is invoked.

environment. A named collection of logical and physical resources used to support the performance of a function.

environment variable. A variable that specifies how an operating system or another program runs, or the devices that the operating system recognizes.

error. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

error log stream. A continuous flow of error information that is transmitted using a predefined format.

ESB. See enterprise service bus.

event.

1. A change to a state, such as the completion or failure of an operation, business process, or human task, that can trigger a subsequent action, such as persisting the event data to a data repository or invoking another business process.

2. A change to data in an enterprise information system (EIS) that is processed by the adapter and used to deliver business objects from the EIS to the endpoints (applications) that need to be notified of the change.

evictor. A component that controls the membership of entries in each BackingMap instance. Sparse caches can use evictors to automatically remove data from the cache without affecting the database.

exception. A condition or event that cannot be handled by a normal process.

exception handler. A set of routines that responds to an abnormal condition. An exception handler is able to interrupt and to resume the normal running of processes.

exclusive lock. A lock that prevents concurrently executing application processes from accessing database data. See also shared lock.

execution trace. A chain of events that is recorded and displayed in a hierarchal format on the Events page of the integration test client.

export. An exposed interface from a Service Component Architecture (SCA) module that offers a business service to the outside world. An export has a binding that defines how the service can be accessed by service requesters, for example, as a Web service.

export file.

1. A file created during the development process for inbound operations that contains the configuration settings for inbound processing.

2. The file containing data that has been exported.

expression. An SQL or XQuery operand or a collection of SQL or XQuery operators and operands that yields a single value.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that is based on Standard Generalized Markup Language (SGML).

eXtreme Scale grid. A pattern that is used to interact with eXtreme Scale when all of the data and clients are in one process.

factory. In object-oriented programming, a class that is used to create instances of another class. A factory is used to isolate the creation of objects of a particular class into one place so that new functions can be provided without widespread code changes.

failover. An automatic operation that switches to a redundant or standby system in the event of a software, hardware, or network interruption.

fine-grained. Pertaining to viewing an individual object in detail.

fire. In object-oriented programming, to cause a state transition.

firewall. A network configuration, typically both hardware and software, that prevents unauthorized traffic into and out of a secure network.

fix pack. A cumulative collection of fixes that is made available between scheduled refresh packs, manufacturing refreshes, or releases. It is intended to allow customers to move to a specific maintenance level. See also interim fix.

folder. A container used to organize objects.

for loop. A loop that repeats the same sequence of activities a specified number of times.

fork. A process element that makes copies of its input and forwards them by several processing paths in parallel.

garbage collection. A routine that searches memory to reclaim space from program segments or inactive data.

General Inter-ORB Protocol (GIOP). A protocol that Common Object Request Broker Architecture (CORBA) uses to define the format of messages.

generic object. An object that is used in API calls and XPATH expressions to refer to concepts, custom entities, or collections. For example, the XPATH expression /WSRR/GenericObject will retrieve all concepts from WebSphere Service Registry and Repository.

getter method. A method whose purpose is to get the value of an instance or class variable. This allows another object to find out the value of one of its variables.

GIOP. See General Inter-ORB Protocol.

global.

1. Pertaining to an element that is available to any process in the workspace. A global element appears in the project tree and can be used in multiple processes. Tasks, processes, repositories, and services can be either global (referenced by any process in the project) or local (specific to a single process).

2. Pertaining to information available to more than one program or subroutine.

global attribute. In XML, an attribute that is declared as a child of the schema element rather than as part of a complex type definition. Global attributes can be referenced in one or more content models using the ref attribute.

global element. In XML, an element that is declared as a child of the schema element rather than as part of a complex type definition. Global elements can be referenced in one or more content models using the ref attribute.

global instance identifier. A globally unique identifier that is generated either by the application or by the emitter and is used as a primary key for event identification.

global security. Pertains to all applications running in the environment and determines whether security is used, the type of registry used for authentication, and other values, many of which act as defaults.

global transaction. A recoverable unit of work performed by one or more resource managers in a distributed transaction environment and coordinated by an external transaction manager.

global variable. A variable that is used to hold and manipulate values assigned to it during translation and that is shared across maps and across document translations. One of the three types of variables supported by the Data Interchange Services mapping command language.

group.

1. A collection of users who can share access authorities for protected resources.

2. A set of related documents within an interchange. An interchange can contain zero to many groups.

3. In places, two or more people who are grouped for membership in a place.

HA. See high availability.

HA group. A collection of one or more members used to provide high availability for a process.

HA policy. A set of rules that is defined for an HA group that dictate whether zero (0), or more members are activated. The policy is associated with a specific HA group by matching the policy match criteria with the group name.

health. The general condition or state of the database environment.

heartbeat. A signal that one entity sends to another to convey that it is still active.

high availability (HA). Pertaining to a clustered system that is reconfigured when node or daemon failures occur, so that workloads can be redistributed to the remaining nodes in the cluster.

high availability manager. A framework within which core group membership is determined and status is communicated between core group members.

host.

1. A computer that is connected to a network and that provides an access point to that network. The host can be a client, a server, or both a client and server simultaneously.

2. In performance profiling, a machine that owns processes that are being profiled. See also server.

host name.

1. In Internet communication, the name given to a computer. The host name might be a fully qualified domain name such as mycomputer.city.company.com, or it might be a specific subname such as mycomputer.

2. The network name for a network adapter on a physical machine in which the node is installed.

host system. An enterprise mainframe computer system that hosts 3270 applications. In the 3270 terminal service development tools, the developer uses the 3270 terminal service recorder to connect to the host system.

HTTP over SSL (HTTPS). A Web protocol for secure transactions that encrypts and decrypts user page requests and pages returned by the Web server.

HTTPS.

1. See HTTP over SSL.

2. See Hypertext Transfer Protocol Secure.

Hypertext Transfer Protocol Secure (HTTPS). An Internet protocol that is used by Web servers and Web browsers to transfer and display hypermedia documents securely across the Internet.

IDE. See integrated development environment.

if-then rule. A rule in which the action (then part) is performed only when the condition (if part) is true.

IIOP. See Internet Inter-ORB Protocol.

import.

1. A development artifact that imports a service that is external to a module.

2. The point at which an SCA module accesses an external service, (a service outside the SCA module) as if it was local. An import defines interactions between the SCA module and the service provider. An import has a binding and one or more interfaces.

index. A set of pointers that are logically ordered by the values of a key. Indexes provide quick access to data and can enforce uniqueness of the key values for the rows in the table.

information center. A collection of information that provides support for users of one or more products, can be launched separately from the product, and includes a list of topics for navigation and a search engine.

inheritance. An object-oriented programming technique in which existing classes are used as a basis for creating other classes. Through inheritance, more specific elements incorporate the structure and behavior of more general elements.

installation package. An installable unit of a software product. Software product packages are separately installable units that can operate independently from other packages of that software product.

installation target. The system on which selected installation packages are installed.

instance. A specific occurrence of an object that belongs to a class.

instantiate. To represent an abstraction with a concrete instance.

integrated development environment (IDE). A set of software development tools, such as source editors, compilers, and debuggers, that are accessible from a single user interface.

interface. A collection of operations that are used to specify a service of a class or a component.

interim fix. A certified fix that is generally available to all customers between regularly scheduled fix packs, refresh packs, or releases. See also fix pack.

Internet Inter-ORB Protocol (IIOP). A protocol used for communication between Common Object Request Broker Architecture (CORBA) object request brokers.

Internet Protocol (IP). A protocol that routes data through a network or interconnected networks. This protocol acts as an intermediary between the higher protocol layers and the physical network.

invocation. The activation of a program or procedure.

IP. See Internet Protocol.

IP sprayer. A device that is located between inbound requests from the users and the application server nodes that reroutes requests across nodes.

iteration. See loop.

iterator. A class or construct that is used to step through a collection of objects one at a time.

JAAS. See Java Authentication and Authorization Service.

JAF. See JavaBeans Activation Framework.

JAR file. A Java archive file. See also Web archive, enterprise archive.

Java. An object-oriented programming language for portable interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated.

Java API for XML (JAX). A set of Java-based APIs for handling various operations involving data defined through Extensible Markup Language (XML).

Java archive. A compressed file format for storing all of the resources that are required to install and run a Java program in a single file. See also Web archive, enterprise archive.

Java Authentication and Authorization Service (JAAS). In Java EE technology, a standard API for performing security-based operations. Through JAAS, services can authenticate and authorize users while enabling the applications to remain independent from underlying technologies.

Java class. A class that is written in the Java language.

Java Command Language. A scripting language for the Java environment that is used to create Web content and to control Java applications.

Java Connector security. An architecture designed to extend the end-to-end security model for Java EE-based applications to include enterprise information systems (EIS).

Java Database Connectivity (JDBC). An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call level interface for SQL-based and XQuery-based database access.

Java EE. See Java Platform, Enterprise Edition.

Java EE application. Any deployable unit of Java EE functionality. This unit can be a single module or a group of modules packaged into an enterprise archive (EAR) file with a Java EE application deployment descriptor. (Sun)

Java EE Connector Architecture (JCA). A standard architecture for connecting the Java EE platform to heterogeneous enterprise information systems (EIS).

Java EE server. A runtime environment that provides EJB or Web containers.

Java file. An editable source file (with .java extension) that can be compiled into bytecode (a .class file).

Java Management Extensions (JMX). A means of doing management of and through Java technology. JMX is a universal, open extension of the Java programming language for management that can be deployed across all industries, wherever management is needed.

Java Message Service (JMS). An application programming interface that provides Java language functions for handling messages.

Java Naming and Directory Interface (JNDI). An extension to the Java platform that provides a standard interface for heterogeneous naming and directory services.

Java platform. A collective term for the Java language for writing programs; a set of APIs, class libraries, and other programs used in developing, compiling, and error-checking programs; and a Java virtual machine which loads and runs the class files. (Sun)

Java Platform, Enterprise Edition (Java EE). An environment for developing and deploying enterprise applications, defined by Sun Microsystems Inc. The Java EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Sun)

Java Platform, Standard Edition (Java SE). The core Java technology platform. (Sun)

Java project. In Eclipse, a project that contains compilable Java source code and is a container for source folders or packages.

Java runtime environment. A subset of a Java developer kit that contains the core executable programs and files that constitute the standard Java platform. The JRE includes the Java virtual machine (JVM), core classes, and supporting files.

Java SE. See Java Platform, Standard Edition.

Java SE Development Kit (JDK). The name of the software development kit that Sun Microsystems provides for the Java platform.

Java Secure Socket Extension (JSSE). A Java package that enables secure Internet communications. It implements a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TSL) protocols and supports data encryption, server authentication, message integrity, and optionally client authentication.

Java Specification Request (JSR). A formally proposed specification for the Java platform.

Java virtual machine (JVM). A software implementation of a processor that runs compiled Java code (applets and applications).

Java virtual machine Profiler Interface (JVMPI). A profiling tool that supports the collection of information, such as data about garbage collection and the Java virtual machine (JVM) API that runs the application server.

JavaBeans. As defined for Java by Sun Microsystems, a portable, platform-independent, reusable component model. See also bean.

JavaBeans Activation Framework (JAF). A standard extension to the Java platform that determines arbitrary data types and available operations and can instantiate a bean to run pertinent services.

Javadoc.

1. A tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields. (Sun)

2. Pertaining to the tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields.

JavaMail API. A platform and protocol-independent framework for building Java-based mail client applications.

JavaScript. A Web scripting language that is used in both browsers and Web servers. (Sun)

JavaScript Object Notation. A lightweight data-interchange format that is based on the object-literal notation of JavaScript. JSON is programming-language neutral but uses conventions from languages that include C, C++, C#, Java, JavaScript, Perl, Python.

JavaServer Pages (JSP). A server-side scripting technology that enables Java code to be dynamically embedded within Web pages (HTML files) and run when the page is served, in order to return dynamic content to a client.

JAX. See Java API for XML.

JCA. See Java EE Connector Architecture.

JDBC. See Java Database Connectivity.

JDK. See Java SE Development Kit.

JMS. See Java Message Service.

JMS data binding. A data binding that provides a mapping between the format used by an external JMS message and the Service Data Object (SDO) representation used by a Service Component Architecture (SCA) module.

JMX. See Java Management Extensions.

JNDI. See Java Naming and Directory Interface.

join.

1. A process element that recombines and synchronizes parallel processing paths after a decision or fork. A join waits for input to arrive at each of its incoming branches before permitting the process to continue.

2. An SQL relational operation in which data can be retrieved from two tables, typically based on a join condition specifying join columns.

3. The configuration on an incoming link that determines the behavior of the link.

JSP. See JavaServer Pages.

JSP file. A scripted HTML file that has a .jsp extension and allows for the inclusion of dynamic content in Web pages. A JSP file can be directly requested as a URL, called by a servlet, or called from within an HTML page.

JSP page. A text-based document using fixed template data and JSP elements that describes how to process a request to create a response. (Sun)

JSR. See Java Specification Request.

JSSE. See Java Secure Socket Extension.

JVM. See Java virtual machine.

JVMPI. See Java virtual machine Profiler Interface.

Jython. An implementation of the Python programming language that is integrated with the Java platform.

key.

1. A cryptographic mathematical value that is used to digitally sign, verify, encrypt, or decrypt a message.

2. Information that characterizes and uniquely identifies the real-world entity that is being tracked by a monitoring context.

keyword. One of the predefined words of a programming language, artificial language, application, or command.

LDAP. See Lightweight Directory Access Protocol.

LDAP directory. A type of repository that stores information on people, organizations, and other resources and that is accessed using the LDAP protocol. The entries in the repository are organized into a hierarchical structure, and in some cases the hierarchical structure reflects the structure or geography of an organization.

library.

1. A collection of model elements, including business items, processes, tasks, resources, and organizations.

2. A project that is used for the development, version management, and organization of shared resources. Only a subset of the artifact types can be created and stored in a library, such as business objects and interfaces.

life cycle. One complete pass through the four phases of software development: inception, elaboration, construction and transition.

Lightweight Directory Access Protocol (LDAP). An open protocol that uses TCP/IP to provide access to directories that support an X.500 model and that does not incur the resource requirements of the more complex X.500 Directory Access Protocol (DAP). For example, LDAP can be used to locate people, organizations, and other resources in an Internet or intranet directory.

Lightweight Third Party Authentication (LTPA). A protocol that uses cryptography to support security in a distributed environment.

listener. A program that detects incoming requests and starts the associated channel.

listener port. An object that defines the association between a connection factory, a destination, and a deployed message-driven bean. Listener ports simplify the administration of the associations between these resources.

load balancing. The monitoring of application servers and management of the workload on servers. If one server exceeds its workload, requests are forwarded to another server with more capacity.

loader. A component that reads data from and writes data to a persistent store.

local.

1. Pertaining to a device, file, or system that is accessed directly from a user system, without the use of a communication line.

2. Pertaining to an element that is available only in its own process.

local database. A database that is located on the workstation in use.

lock. A means of preventing uncommitted changes made by one application process from being perceived by another application process and for preventing one application process from updating data that is being accessed by another process. A lock ensures the integrity of data by preventing concurrent users from accessing inconsistent data.

logging. The recording of data about specific events on the system, such as errors.

long name. The property that specifies the logical name for the server on the z/OS® platform.

loop. A sequence of instructions performed repeatedly.

LTPA. See Lightweight Third Party Authentication.

maintenance mode. A state of a node or server that an administrator can use to diagnose, maintain, or tune the node or server without disrupting incoming traffic in a production environment.

Managed Bean (MBean). In the Java Management Extensions (JMX) specification, the Java objects that implement resources and their instrumentation.

map.

1. A data structure that maps keys to values.

2. A file that defines the transformation between sources and targets.

3. In the EJB development environment, the specification of how the container-managed persistent fields of an enterprise bean correspond to columns in a relational database table or other persistent storage.

MBean. See Managed Bean.

MBean provider. A library containing an implementation of a Java Management Extensions (JMX) MBean and its MBean Extensible Markup Language (XML) descriptor file.

memory leak. The effect of a program that maintains references to objects that are no longer required and therefore need to be reclaimed.

method. In object-oriented programming, an operation that an object can perform. An object can have many methods.

metric. A holder for information, typically a business performance measurement, in a monitoring context.

namespace. A logical container in which all the names are unique. The unique identifier for an artifact is composed of the namespace and the local name of the artifact.

node.

1. A logical grouping of managed servers.

2. Any item on a tree control, including a simple element, compound element, mapping command, comment, or group node.

3. In XML, the smallest unit of valid, complete structure in a document.

4. The fundamental shapes that make up a diagram.

node agent. An administrative agent that manages all application servers on a node and represents the node in the management cell.

object. In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data.

Object Request Broker (ORB). In object-oriented programming, software that serves as an intermediary by transparently enabling objects to exchange requests and responses.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished but instead on what data objects comprise the problem and how they are manipulated.

ObjectGrid. A grid-enabled, memory database for applications that are written in Java. ObjectGrid can be used as an in-memory database or to distribute data across a network.

ODBC. See Open Database Connectivity.

Open Database Connectivity (ODBC). A standard application programming interface (API) for accessing data in both relational and nonrelational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface.

open source. Pertaining to software whose source code is publicly available for use or modification. Open source software is typically developed as a public collaboration and made freely available, although its use and redistribution might be subject to licensing restrictions. Linux is a well known example of open source software.

operation. An implementation of functions or queries that an object might be called to perform.

ORB. See Object Request Broker.

organization. An entity where people cooperate to accomplish specified objectives, such as an enterprise, a company, or a factory.

package.

1. In Java programming, a group of types. Packages are declared with the package keyword. (Sun)

2. The wrapper around the document content that defines the format used to transmit a document over the Internet, for example, RNIF, AS1, and AS2.

3. To assemble components into modules and modules into enterprise applications.

partitioning facility. A programming framework and a system management infrastructure that supports the concept of partitioning for enterprise beans, HTTP traffic, and database access.

Performance Monitoring Infrastructure (PMI). A set of packages and libraries assigned to gather, deliver, process, and display performance data.

permission. Authorization to perform activities, such as reading and writing local files, creating network connections, and loading native code.

persist. To be maintained across session boundaries, typically in nonvolatile storage such as a database system or a directory.

persistence.

1. A characteristic of data that is maintained across session boundaries, or of an object that continues to exist after the execution of the program or process that created it, typically in nonvolatile storage such as a database system.

2. In Java EE, the protocol for transferring the state of an entity bean between its instance variables and an underlying database. (Sun)

persistent data store. A nonvolatile storage for event data, such as a database system, that is maintained across session boundaries and that continues to exist after the execution of the program or process that created it.

pessimistic locking. A locking strategy whereby a lock is held between the time that a row is selected and the time that a searched update or delete operation is attempted on that row.

plug-in. A separately installable software module that adds function to an existing program, application, or interface.

PMI. See Performance Monitoring Infrastructure.

point-to-point. Pertaining to a style of messaging application in which the sending application knows the destination of the message.

policy. A set of considerations that influence the behavior of a managed resource or a user.

port. As defined in a Web Services Description Language (WSDL) document, a single endpoint that is defined as a combination of a binding and a network address.

port number. In Internet communications, the identifier for a logical connector between an application entity and the transport service.

primary key.

1. An object that uniquely identifies an entity bean of a particular type.

2. In a relational database, a key that uniquely identifies one row of a database table.

primitive type. In Java, a category of data type that describes a variable that contains a single value of the appropriate size and format for its type: a number, a character, or a Boolean value. Examples of primitive types include byte, short, int, long, float, double, char, boolean.

process.

1. A progressively continuing procedure consisting of a series of controlled activities that are systematically directed toward a particular result or end.

2. The sequence of documents or messages to be exchanged between the Community Managers and participants to run a business transaction.

profile. Data that describes the characteristics of a user, group, resource, program, device, or remote location.

program temporary fix (PTF). For System i®, System p®, and System z® products, a fix that is tested by IBM and is made available to all customers. See also fix pack.

prompt. A component of an action that indicates that user input is required for a field before making a transition to an output screen.

property. A characteristic of an object that describes the object. A property can be changed or modified. Properties can describe an object name, type, value, or behavior, among other things.

protocol binding. A binding that enables the enterprise service bus to process messages independently of the communication protocol.

proxy. An application gateway from one network to another for a specific network application such as Telnet or FTP, for example, where a firewall proxy Telnet server performs authentication of the user and then lets the traffic flow through the proxy as if it were not there. Function is performed in the firewall and not in the client workstation, causing more load in the firewall.

proxy cluster. A group of proxy servers that distributes HTTP requests across the cluster.

proxy peer access point. A means of identifying the communication settings for a peer access point that cannot be accessed directly.

proxy server.

1. A server that acts as an intermediary for HTTP Web requests that are hosted by an application or a Web server. A proxy server acts as a surrogate for the content servers in the enterprise.

2. A server that receives requests intended for another server and that acts on behalf of the client (as the client's proxy) to obtain the requested service. A proxy server is often used when the client and the server are incompatible for direct connection. For example, the client is unable to meet the security authentication requirements of the server but should be permitted some services.

PTF. See program temporary fix.

public.

1. In object-oriented programming, pertaining to a class member that is accessible to all classes.

2. In the Java programming language, pertains to a method or variable that can be accessed by elements residing in other classes. (Sun)

QoS. See quality of service.

qualifier. A simple element that gives another generic compound or simple element a specific meaning. Qualifiers are used in mapping single or multiple occurrences. A qualifier can also be used to denote the namespace used to interpret the second part of the name, typically referred to as the ID.

quality of service (QoS). A set of communication characteristics that an application requires. Quality of Service (QoS) defines a specific transmission priority, level of route reliability, and security level.

query.

1. A request for information from a database based on specific conditions: for example, a request for a list of all customers in a customer table whose balances are greater than USD1000.

2. A reusable request for information about one or more model elements

read-through cache. A sparse cache that loads data entries by key as they are requested. When data cannot be found in the cache, the missing data is retrieved with the loader, which loads the data from the back-end data repository and inserts the data into the cache.

recursion. A programming technique in which a program or routine calls itself to perform successive steps in an operation, with each step using the output of the preceding step.

refresh pack. A cumulative collection of fixes that contains new functions. See also fix pack, interim fix.

region. A contiguous area of virtual storage that has common characteristics and that can be shared between processes.

replica. A server that contains a copy of the directory or directories of another server. Replicas back up servers in order to enhance performance or response times and to ensure data integrity.

replication. The process of maintaining a defined set of data in more than one location. Replication involves copying designated changes for one location (a source) to another (a target) and synchronizing the data in both locations.

resource.

1. A discrete asset, for example application suites, applications, business services, interfaces, endpoints, and business events.

2. A facility of a computing system or operating system required by a job, task, or running program. Resources include main storage, input/output devices, the processing unit, data sets, files, libraries, folders, application servers, and control or processing programs.

3. A person, piece of equipment, or material that is used to perform a task or a project. Each resource is a particular occurrence or example of a resource definition.

role.

1. A description of a function to be carried out by an individual or bulk resource, and the qualifications required to fulfill the function. In simulation and analysis, the term role is also used to refer to the qualified resources.

2. A job function that identifies the tasks that a user can perform and the resources to which a user has access. A user can be assigned one or more roles.

3. A logical group of principals that provides a set of permissions. Access to operations is controlled by granting access to a role.

4. In a relationship, a role determines the function and participation of entities. Roles capture structure and constraint requirements on participating entities and their manner of participation. For example, in an employment relationship, the roles are employer and employee.

root. The user name for the system user with the most authority.

run time. The time period during which a computer program is running.

runtime topology. A depiction of the momentary state of the environment.

scalability. The ability of a system to expand as resources, such as processors, memory, or storage, are added.

scope.

1. A specification of the boundary within which system resources can be used.

2. In Web services, a property that identifies the lifetime of the object serving the invocation request.

script. A series of commands, combined in a file, that carry out a particular function when the file is run. Scripts are interpreted as they are run.

scripting. A style of programming that reuses existing components as a base for building applications.

SDK. See software development kit.

Secure Sockets Layer (SSL). A security protocol that provides communication privacy. With SSL, client/server applications can communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery.

security administrator. The person who controls access to business data and program functions.

security token. A representation of a set of claims that are made by a client that can include a name, password, identity, key, certificate, group, privilege, and so on.

serialization. In object-oriented programming, the writing of data in sequential fashion to a communications medium from program memory.

serializer. A method for converting object data to another form such as binary or XML.

servant region. A contiguous area of virtual storage that is dynamically started as load increases and automatically stopped as load eases.

server. A software program or a computer that provides services to other software programs or other computers. See also host.

server cluster. A group of servers that are typically on different physical machines and have the same applications configured within them, but operate as a single logical server.

service level agreement (SLA). A contract between a customer and a service provider that specifies the expectations for the level of service with respect to availability, performance, and other measurable objectives.

servlet. A Java program that runs on a Web server and extends the server functions by generating dynamic content in response to Web client requests. Servlets are commonly used to connect databases to the Web.

session.

1. A logical or virtual connection between two stations, software programs, or devices on a network that allows the two elements to communicate and exchange data.

2. A series of requests to a servlet originating from the same user at the same browser.

3. In Java EE, an object used by a servlet to track user interaction with a Web application across multiple HTTP requests.

session affinity. A method of configuring applications in which a client is always connected to the same server. These configurations disable workload management after an initial connection by forcing a client request to always go to the same server.

setter method. A method whose purpose is to set the value of an instance or class variable. This capability allows another object to set the value of one of its variables.

shard. An instance of a partition. A shard can be a primary or replica.

shared lock. A lock that limits concurrently running application processes to read-only operations on database data.

shell script. A program, or script, that is interpreted by the shell of an operating system.

signer certificate. The trusted certificate entry that is typically in a truststore file.

silent installation. An installation that does not send messages to the console but instead stores messages and errors in log files. A silent installation can use response files for data input.

silent mode. A method for installing or uninstalling a product component from the command line with no GUI display. When using silent mode, you specify the data required by the installation or uninstallation program directly on the command line or in a file (called an option file or response file).

skeleton. Scaffolding for an implementation class.

SLA. See service level agreement.

software development kit (SDK). A set of tools, APIs, and documentation to assist with the development of software in a specific computer language or for a particular operating environment.

SQL. See Structured Query Language.

SQL query. A component of certain SQL statements that specifies a result table.

SSL. See Secure Sockets Layer.

SSL channel. A type of channel within a transport chain that associates a Secure Sockets Layer (SSL) configuration repertoire with the transport chain.

stack. An area in memory that typically stores information such as temporary register information, values of parameters, and return addresses of subroutines and is based on the principle of last in, first out (LIFO).

stand-alone. Independent of any other device, program, or system. In a network environment, a stand-alone machine accesses all required resources locally.

stand-alone server. A catalog service or container server that is managed from the operating system that starts and stops the server process.

static. A Java programming language keyword that is used to define a variable as a class variable.

string. In programming languages, the form of data used for storing and manipulating text.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

subclass. In Java, a class that is derived from a particular class, through inheritance.

subquery. In SQL, a subselect used within a predicate, for example, a select-statement within the WHERE or HAVING clause of another SQL statement.

synchronize. To add, subtract, or change one feature or artifact to match another.

synchronous process. A process that starts by invoking a request-response operation. The result of the process is returned by the same operation.

synchronous replica. A shard that receives updates as part of the transaction on the primary shard to guarantee data consistency, which can increase the response time compared with an asynchronous replica.

syntax. The rules for the construction of a command or statement.

systems analyst. A specialist who is responsible for translating business requirements into system definitions and solutions.

TCP. See Transmission Control Protocol.

TCP channel. A type of channel within a transport chain that provides client applications with persistent connections within a local area network (LAN).

TCP/IP. See Transmission Control Protocol/Internet Protocol.

TCP/IP monitoring server. A runtime environment that monitors all requests and responses between a Web browser and an application server, as well as TCP/IP activity.

thin application client. A lightweight, downloadable Java application run time capable of interacting with enterprise beans.

thin client. A client that has little or no installed software but has access to software that is managed and delivered by network servers that are attached to it. A thin client is an alternative to a full-function client such as a workstation.

thread. A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

thread contention. A condition in which a thread is waiting for a lock or object that another thread holds.

threshold. A setting that applies to an interrupt in a simulation that defines when a process simulation should be halted based on a condition existing for a specified proportion of occurrences of some event.

throughput. The measure of the amount of work performed by a device, such as a computer or printer, over a period of time, for example, number of jobs per day.

time to live. The time interval in seconds that an entry can exist in the cache before that entry is discarded.

timeout. A time interval that is allotted for an event to occur or complete before operation is interrupted.

timer. A task that produces output at certain points in time.

timing constraint. A specialized validation action used to measure the duration of a method call or a sequence of method calls.

Tivoli Performance Viewer. A Java client that retrieves the Performance Monitoring Infrastructure (PMI) data from an application server and displays it in various formats.

token.

1. A marker used to track the current state of a process instance during a simulation run.

2. A particular message or bit pattern that signifies permission or temporary control to transmit over a network.

topology. The physical or logical mapping of the location of networking components or nodes within a network. Common network topologies include bus, ring, star, and tree.

transaction. A process in which all of the data modifications that are made during a transaction are either committed together as a unit or rolled back as a unit.

Transmission Control Protocol (TCP). A communication protocol used in the Internet and in any network that follows the Internet Engineering Task Force (IETF) standards for internetwork protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in interconnected systems of such networks.

Transmission Control Protocol/Internet Protocol (TCP/IP). An industry-standard, nonproprietary set of communication protocols that provides reliable end-to-end connections between applications over interconnected networks of different types.

truststore file. A key database file that contains the public keys for a trusted entity.

type.

1. In Java programming, a class or interface.

2. In a WSDL document, an element that contains data type definitions using some type system (such as XSD).

UDDI. See Universal Description, Discovery, and Integration.

Uniform Resource Identifier (URI).

1. A compact string of characters for identifying an abstract or physical resource.

2. A unique address that is used to identify content on the Web, such as a page of text, a video or sound clip, a still or animated image, or a program. The most common form of URI is the Web page address, which is a particular form or subset of URI called a Uniform Resource Locator (URL). A URI typically describes how to access the resource, the computer that contains the resource, and the name of the resource (a file name) on the computer.

Uniform Resource Locator (URL). The unique address of an information resource that is accessible in a network such as the Internet. The URL includes the abbreviated name of the protocol used to access the information resource and the information used by the protocol to locate the information resource.

Uniform Resource Name (URN). A name that uniquely identifies a Web service to a client.

Universal Description, Discovery, and Integration (UDDI). A set of standards-based specifications that enables companies and applications to quickly and easily find and use Web services over the Internet.

Universally Unique Identifier (UUID). The 128-bit numerical identifier that is used to ensure that two components do not have the same identifier.

UNIX System Services. An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specifications and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

upgradeable lock. A lock that identifies the intent to update a cache entry when using a pessimistic lock.

upstream. Pertaining to the direction of the flow, which is from the start of the process (upstream) toward the end of the process (downstream).

URI. See Uniform Resource Identifier.

URL. See Uniform Resource Locator.

URL scheme. A format that contains another object reference.

URN. See Uniform Resource Name.

UUID. See Universally Unique Identifier.

variable. A representation of a changeable value.

version. A separately licensed program that typically has significant new code or new function.

virtual host. A configuration enabling a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine.

virtual machine. An abstract specification for a computing device that can be implemented in different ways in software and hardware.

virtualization. A technique that encapsulates the characteristics of resources from the way in which other systems interact with those resources.

waiter. A thread waiting for a connection.

WAR. See Web archive.

WCCM. See WebSphere Common Configuration Model.

Web archive (WAR). A compressed file format, defined by the Java EE standard, for storing all the resources required to install and run a Web application in a single file. See also enterprise archive.

Web browser. A client program that initiates requests to a Web server and displays the information that the server returns.

Web component. A servlet, JavaServer Pages (JSP) file, or a HyperText Markup Language (HTML) file. One or more Web components make up a Web module.

Web container. A container that implements the Web component contract of the Java EE architecture. (Sun)

Web container channel. A type of channel within a transport chain that creates a bridge in the transport chain between an HTTP inbound channel and a servlet or JavaServer Pages (JSP) engine.

Web crawler. A type of crawler that explores the Web by retrieving a Web document and following the links within that document.

Web server. A software program that is capable of servicing Hypertext Transfer Protocol (HTTP) requests.

Web server plug-in. A software module that supports the Web server in communicating requests for dynamic content, such as servlets, to the application server.

Web server separation. A topology where the Web server is physically separated from the application server.

Web site. A related collection of files available on the Web that is managed by a single entity (an organization or an individual) and contains information in hypertext for its users. A Web site often includes hypertext links to other Web sites.

WebSphere. An IBM brand name that encompasses tools for developing e-business applications and middleware for running Web applications.

WebSphere Common Configuration Model (WCCM). A model that provides for programmatic access to configuration data.

what you see is what you get (WYSIWYG). A capability of an editor to continually display pages exactly as they will be printed or otherwise rendered.

while loop. A loop that repeats the same sequence of activities as long as some condition is satisfied. The while loop tests its condition at the beginning of every loop. If the condition is false from the start, the sequence of activities contained in the loop never runs.

WLM. See Workload Manager.

workload management. The optimization of the distribution of incoming work requests to the application servers, enterprise beans, servlets and other objects that can effectively process the request.

Workload Manager (WLM). A component of z/OS that provides the ability to run multiple workloads at the same time within one z/OS image or across multiple images.

workspace.

1. A directory on disk that contains all project files, as well as information such as preferences.

2. A temporary repository of configuration information that administrative clients use.

3. In Eclipse, the collection of projects and other resources that the user is currently developing in the workbench. Metadata about these resources resides in a directory on the file system; the resources might reside in the same directory.

write-behind cache. A cache that asynchronously writes each write operation to the database using a loader.

write-through cache. A cache that synchronously writes each write operation to the database using a loader.

WYSIWYG. See what you see is what you get.

X/Open XA. The X/Open Distributed Transaction Processing XA interface. A proposed standard for distributed transaction communication. The standard specifies a bidirectional interface between resource managers that provide access to shared resources within transactions, and between a transaction service that monitors and resolves transactions.

XA. A bidirectional interface between one or more resource managers that provide access to shared resources and a transaction manager that monitors and resolves transactions.

XML. See Extensible Markup Language.

z/OS. An IBM mainframe operating system that uses 64-bit real storage.

zone-based support. A function that enables rules-based shard placement to improve grid availability by placing shards across different data centers, whether on different floors or even in different buildings or geographies.

Notices

Trademarks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX®
- CICS®
- Cloudscape
- DB2
- Domino®
- IBM
- Lotus®
- RACF®
- Redbooks®
- Tivoli
- WebSphere
- z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

LINUX is a trademark of Linus Torvalds in the U.S., other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Index

Printed in USA