IBM WebSphere eXtreme Scale Version 7.1

Product Overview

June 15, 2011

IBM

This edition applies to version 7, release 1, of WebSphere eXtreme Scale and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About the *Product Overview*

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 *Product Overview* contains a high-level view of WebSphere eXtreme Scale concepts, including use case scenarios, and tutorials.
- The *Installation Guide* describes how to install common topologies of WebSphere eXtreme Scale.
- 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.

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 for anyone that is interested in learning about WebSphere eXtreme Scale.

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. WebSphere eXtreme Scale overview

WebSphere eXtreme Scale is an elastic, scalable, in-memory data grid. The data grid dynamically caches, partitions, replicates, and manages application data and business logic across multiple servers. WebSphere eXtreme Scale performs massive volumes of transaction processing with high efficiency and linear scalability. With WebSphere eXtreme Scale, you can also get qualities of service such as transactional integrity, high availability, and predictable response times.

WebSphere eXtreme Scale can be used in different ways. You can use the product as a very powerful cache, as an in-memory database processing space to manage application state, or to build Extreme Transaction Processing (XTP) applications. These XTP capabilities include an application infrastructure to support your most demanding business-critical applications.

Elastic scalability

Elastic scalability is possible through the use of distributed object caching. With elastic scalability, the data grid monitors and manages itself. The data grid can add or remove servers from the topology, which increases or decreases memory, network throughput, and processing capacity as needed. When a scale-out process is initiated, capacity is added to the data grid while it is running without requiring a restart. Conversely, a scale-in process immediately removes capacity. The data grid is also self-healing by automatically recovering from failures.

WebSphere eXtreme Scale versus an in-memory database

WebSphere eXtreme Scale cannot be considered an actual in-memory database. An in-memory database is too simple to handle some of the complexities that WebSphere eXtreme Scale can manage. If an in-memory database has a server that fails, it cannot repair the issue. A failure can be disastrous if your entire environment is on that one server.

To tackle the problem of this type of failure, eXtreme Scale splits the given data set into partitions, which are equivalent to constrained tree schemas. Constrained tree schemas describe the relationship between entities. When you are using partitions, the entity relationships must model a tree data structure. In this structure, the head of the tree is the root entity and is the only entity that is partitioned. All other children of the root entity are stored in the same partition as the root entity. Each partition exists as a primary copy, or shard. A partition also contains replica shards for backing up the data. An in-memory database cannot provide this function because it is not structured and dynamic in this way. With an in-memory database, you must implement the operations that WebSphere eXtreme Scale does automatically. You can run SQL operations on in-memory databases, improving the processing speed compared to databases that are not in memory. WebSphere eXtreme Scale has its own query language instead of SQL support. This query language is more elastic, enables partitioning of data, and provides dependable failure recovery.

WebSphere eXtreme Scale with databases

With the write-behind cache feature, WebSphere eXtreme Scale can serve as a front-end cache for a database. By using this front-end cache, throughput increases while reducing database load and contention. WebSphere eXtreme Scale provides predictable scaling in and scaling out at predictable processing cost.

The following image shows that in a distributed, coherent cache environment, the eXtreme Scale clients send and receive data from the data grid. The data grid can be automatically synchronized with a backend data store. The cache is coherent because all of the clients see the same data in the cache. Each piece of data is stored on exactly one writable server in the cache. Having one copy of each piece of data prevents wasteful copies of records that might contain different versions of the data. A coherent cache holds more data as more servers are added to the data grid, and scales linearly as the data grid grows in size. The data can also be optionally replicated for additional fault tolerance.

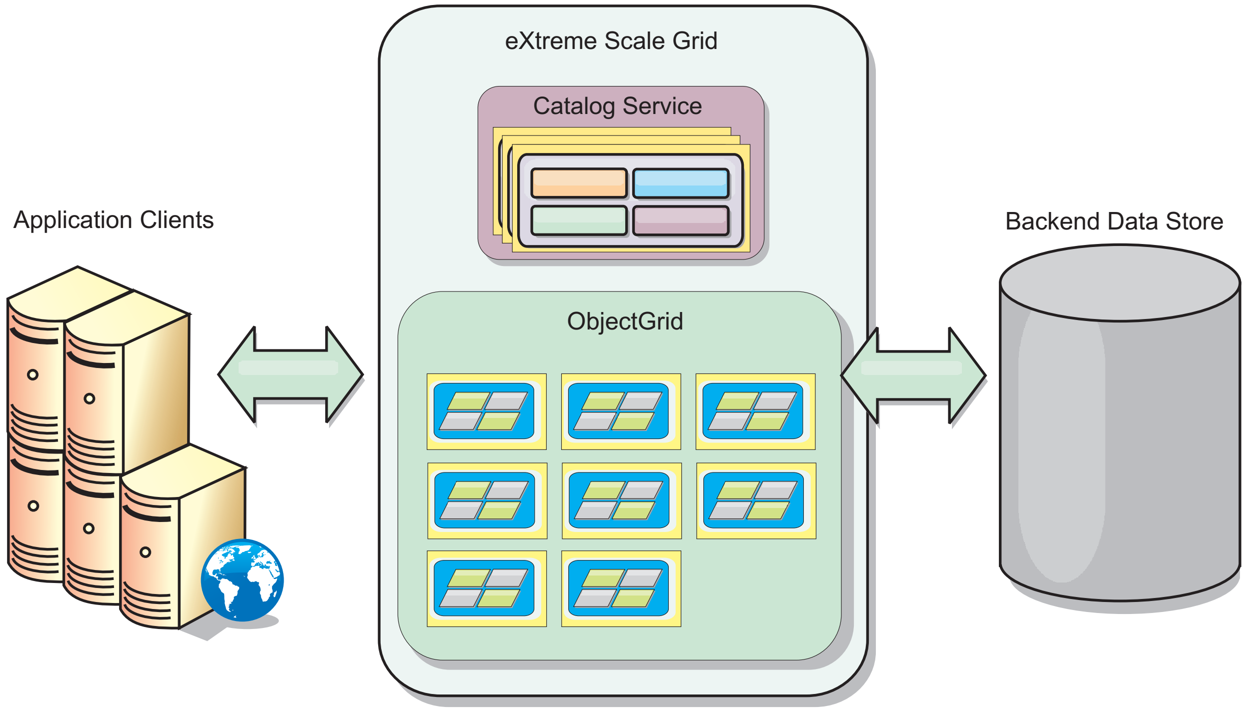


Figure 1. High-level topology

WebSphere eXtreme Scale has servers, called *container servers*, that provide its in-memory data grid. These servers can run inside WebSphere Application Server, or on simple Java Standard Edition (J2SE) Java virtual machines. More than one container server can run on a single physical server. As a result, the in-memory data grid can be large. The data grid is not limited by, and does not have an impact on, the memory or address space of the application or the application server. The memory can be the sum of the memory of several hundred, or thousand, Java virtual machines, running on many different physical servers.

As an in-memory database processing space, WebSphere eXtreme Scale can be backed by disk, database, or both.

While eXtreme Scale provides several Java APIs, many use cases require no user programming, just configuration and deployment in your WebSphere infrastructure.

Data grid overview

The fundamental data grid paradigm is a key-value pair, where the data grid stores values (Java objects), with an associated key (another Java object). The key is later used to retrieve the value. In eXtreme Scale, a map consists of entries of such key-value pairs.

WebSphere eXtreme Scale offers a number of data grid configurations, from a single, simple local cache, to a large distributed cache, using multiple Java virtual machines or servers.

In addition to storing simple Java objects, you can store objects with relationships. You can use a query language that is like SQL, with `SELECT ... FROM ... WHERE` statements to retrieve these objects. For example, an order object might have a customer object and multiple item objects associated with it. WebSphere eXtreme Scale supports one-to-one, one-to-many, many-to-one, and many-to-many relationships.

WebSphere eXtreme Scale also supports an EntityManager programming interface for storing entities in the cache. This programming interface is like entities in Java Enterprise Edition. Entity relationships can be automatically discovered from an entity descriptor XML file or annotations in the Java classes. You can retrieve an entity from the cache by primary key using the find method on the EntityManager interface. Entities can be persisted to or removed from the data grid within a transaction boundary.

Consider a distributed example where the key is a simple alphabetic name. The cache might be split into four partitions by key: partition 1 for keys starting with A-E, partition 2 for keys starting with F-L, and so on. For availability, a partition has a primary shard and a replica shard. Changes to the cache data are made to the primary shard, and replicated to the replica shard. You configure the number of servers that contain the data grid data, and eXtreme Scale distributes the data into shards over these server instances. For availability, replica shards are placed in separate physical servers from primary shards.

WebSphere eXtreme Scale uses a catalog service to locate the primary shard for each key. It handles moving shards among eXtreme Scale servers when the physical servers fail and later recover. For example, if the server containing a replica shard fails, eXtreme Scale allocates a new replica shard. If a server containing a primary shard fails, the replica shard is promoted to be the primary shard. As before, a new replica shard is constructed.

The simplest eXtreme Scale programming interface is the ObjectMap interface, which is a simple map interface that includes: a map.put(key,value) method to put a value in the cache, and a map.get(key) method to later retrieve the value.

What is new in this release

WebSphere eXtreme Scale includes many new features in Version 7.1, including integration with the dynamic cache, byte array maps, and more.

7.1.0.3+

What is new in WebSphere eXtreme Scale Version 7.1.0.3

Feature	Description
URL rewriting support for HTTP session persistence	You can now persist sessions that use URL rewriting as a session tracking mechanism. Set the **`useURLEncoding`** property to `true` in the `splicer.properties` file to enable tracking for sessions that use URL rewriting. See the information about configuring HTTP session managers in the *Administration Guide* for more information.

7.1.0.2+

What is new in WebSphere eXtreme Scale Version 7.1.0.2

Feature	Description
Byte array maps are supported in multi-master topologies	Maps that are configured with COPY_TO_BYTES copy mode can replicate across catalog service domains. See the initial considerations for multi-master topologies in the *Administration Guide* for more information about updating your multi-master topology to use COPY_TO_BYTES copy mode.
New **`-triggerPlacement`** and **`-placementStatus`** parameters in the **`xsadmin`** utility	You can use the new **`-triggerPlacement`** parameter in the **`xsadmin`** utility to force placement to occur. Forcing placement to occur can be useful when you are completing maintenance tasks on the container servers. See the information about forcing placement to occur in the *Administration Guide* for more information. You can also use the **`-placementStatus`** parameter in the **`xsadmin`** utility to display the current configured and runtime placement for your configuration. See **`xsadmin`** utility reference in the *Administration Guide* for more information.

Fix 1+

What is new in WebSphere eXtreme Scale Version 7.1 Fix 1

Feature	Description
Create configuration profiles for the **`xsadmin`** utility	You can save a configuration profile for the **`xsadmin`** utility so that your **`xsadmin`** utility calls are shorter. You can save parameters such as the user name, password, and other security options. See the information about creating a configuration profile in the xsadmin utility for more information.

What is new in WebSphere eXtreme Scale Version 7.1

Table 1. New features in WebSphere eXtreme Scale Version 7.1

Feature	Description
DB2® client information integration	Integrate eXtreme Scale JPA Loader plug-ins with DB2, so when DB2 is used as the backend database, the WebSphere eXtreme Scale information (username, workstation name, application name, and accounting information) can be made available in the DB2 Performance Monitor. This feature allows enabling and disabling configuration of client info to DB2. This function is disabled by default. For more information, see the information about monitoring eXtreme Scale information in DB2 in the *Administration Guide* for more information.
Catalog service domain configuration	Catalog service domains can be configured using the WebSphere Application Server administrative console or using administrative tasks. Catalog service domains define a group. For more information, see the information on creating catalog service domains in the *Administration Guide*.
Multi-master replication	Multiple data centers can be linked together asynchronously, allowing data center to access data locally and maintain high availability. See Multi-master data grid replication topologies (AP) for more information.
Last update time TTL evictor	The TTL evictor has been updated to track the time in which an entry was updated, expanding on the time-to-live evictor. For more information, see the information about the TimeToLive (TTL) evictor in the *Programming Guide*
usedBytes statistics for in-memory grids	The amount of memory that is used by cache entries in a BackingMap can be tracked using all of the statistics providers. For more information, see the information about cache memory consumption sizing in the *Administration Guide*.
Dynamic statistics	Statistics can now be enabled and disabled on demand. For more information, see the information about monitoring with MBeans in the *Administration Guide*.
Monitoring console	The graphical monitoring console provides current and historical views into WebSphere eXtreme Scale server statistics. For more information, see the information about the web console in the *Administration Guide*.
Improved HTTP Session Manager	Configuration of the HTTP session manager has been simplified. You can now configure the HTTP session manager in the WebSphere Application Server administrative console. For more information, see the information about configuring the HTTP session manager in the *Administration Guide*.
Multi-homed client support	Clients can be configured to use a specific network adapter. For more information, see the information about the client properties file in the *Administration Guide*.

Table 1. New features in WebSphere eXtreme Scale Version 7.1 (continued)

Feature	Description
ISA Lite	IBM® Support Assistant Lite for WebSphere eXtreme Scale provides automatic data collection and symptom analysis support for problem determination scenarios. For more information, see the information about the IBM support assistant for WebSphere eXtreme Scale in the *Administration Guide*.
REST	The REST data service provides non-Java clients access to eXtreme Scale data, supporting the Open Data Protocol (OData), providing full compatibility with Microsoft WCF Data Services. For more information, see Chapter 8, "REST data services overview," on page 155.
Client-only installation	WebSphere eXtreme Scale clients can now be installed independently, decreasing the installation footprint for WebSphere eXtreme Scale applications. For more information, see the information about installing and deploying WebSphere eXtreme Scale in the *Administration Guide*.
Deprecated features	For a list of deprecated properties and APIs, see the deprecated items in the *Administration Guide*.

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. Clicking the links provided here will result in a search of the Support Web page for the relevant release notes, which will be returned as a list.

- **7.1+** To see a list of the release notes for Version 7.1, 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.

WebSphere eXtreme Scale technical overview

WebSphere eXtreme Scale is an elastic, scalable, in-memory data grid. It dynamically caches, partitions, replicates, and manages application data and business logic across multiple servers.

Because WebSphere eXtreme Scale is not an in-memory database, you must consider specific configuration requirements. The first step to deploying a data grid is to start a core group and catalog service, which acts as coordinator for all other Java virtual machines that are participating in the data grid and manage configuration information. WebSphere eXtreme Scale processes are started with simple command script invocations from the command line.

The next step is to start WebSphere eXtreme Scale server processes for the data grid to store and retrieve data. As servers are started, they automatically register themselves with the core group and catalog service allowing them to cooperate in providing data grid services. More servers increase both data grid capacity and reliability.

A local data grid is a simple, single-instance grid where all the data is in the one grid. To effectively use WebSphere eXtreme Scale as an in-memory database processing space, you can configure and deploy a distributed data grid. The data in the distributed grid is spread out over the various eXtreme Scale servers containing it such that each server contains only some of the data, called a partition.

A key distributed data grid configuration parameter is the number of partitions in the grid. The grid data is partitioned into this number of subsets, each of which is called a partition. The catalog service locates the partition for a given datum based on its key. The number of partitions directly affects the capacity and scalability of the data grid. A server cancontain one or more data grid partitions. Thus the server's memory space limits the size of a partition. Conversely, increasing the number of partitions increases the capacity of the data grid. The maximum capacity of a data grid is the number of partitions times the usable memory size of each server. A server can be a JVM, but you can define your eXtreme Scale server to suit your deployment environment.

The data of a partition is stored in a shard. For availability, a data grid can be configured with replicas, which can be synchronous or asynchronous. Changes to the grid data are made to the primary shard, and replicated to the replica shards. The total memory that is used or required by a data grid is thus the size of the data grid times (1 (for the primary) + the number of replicas).

WebSphere eXtreme Scale distributes the shards of a data grid over the number of servers comprising the grid. These servers may be on the same or different physical servers. For availability, replica shards are placed in separate physical servers from primary shards.

WebSphere eXtreme Scale monitors the status of its servers and moves shards during shard or physical server failure and recovery. For example, if the server containing a replica shard fails, WebSphere eXtreme Scale allocates a new replica shard, and replicate data from the primary to the new replica. If a server that contains a primary shard faisl, the replica shard is promoted to be the primary shard, and, a new replica shard is constructed. If you start an additional server for the data grid, the shards are balanced over all servers. This rebalancing is called scale-out. Similarly, for scale-in, you might stop one of the servers to reduce the resources that are used by a data grid. As a result, the shards are balanced over the remaining servers.

Planning overview

Before using WebSphere eXtreme Scale in a production environment, consider the following issues to optimize your deployment.

Installation considerations

You can install WebSphere eXtreme Scale in a stand-alone environment, or you can integrate the installation with WebSphere Application Server. To ensure that you are able to seamlessly upgrade your servers in the future, you must plan your environment accordingly. For the best performance, catalog servers should run on different machines than the container servers. If you must run your catalog servers and container servers on the same machine, then use separate installations of WebSphere eXtreme Scale for the catalog and container servers. By using two installations, you can upgrade the installation that is running the catalog server first. See the information about updating eXtreme Scale servers in the *Administration Guide* for more information.

Caching topology considerations

Your architecture can use local in-memory data caching or distributed client-server data caching. Each type of cache topology has advantages and disadvantages. The caching topology you implement depends on the requirements of your environment and application. For more information about the different caching topologies, see “Caching topology: In-memory and distributed caching” on page 16.

Data capacity considerations

The following list includes items to consider:

- **Number of systems and processors**: How many physical machines and processors are needed in the environment?
- **Number of servers**: How many eXtreme Scale servers to host eXtreme Scale maps?
- **Number of partitions**: The amount of data stored in the maps is one factor in determining the number of partitions needed.
- **Number of replicas**: How many replicas are required for each primary in the domain?
- **Synchronous or asynchronous replication**: Is the data vital so that synchronous replication is required? Or is performance a higher priority, making asynchronous replication the correct choice?
- **Heap sizes**: How much data will be stored on each server?

the information about capacity planning in the *Administration Guide*.

Integrate with WebSphere products

You can integrate WebSphere eXtreme Scale with other server products, such as WebSphere Application Server and WebSphere Application Server Community Edition.

WebSphere Application Server Community Edition

WebSphere Application Server Community Edition can share session state, but not in an efficient, scalable manner. WebSphere eXtreme Scale provides a high performance, distributed persistence layer that can be used to replicate state, but does not readily integrate with any application server outside of WebSphere Application Server. You can integrate these two products to provide a scalable session-management solution.

WebSphere Application Server

You can integrate WebSphere Application Server into various aspects of your WebSphere eXtreme Scale configuration. You can deploy data grid applications and use WebSphere Application Server to host container and catalog servers. You can also use WebSphere Application Server security in your WebSphere eXtreme Scale environment. See the following topics for more information:

- For information about configuring WebSphere eXtreme Scale withWebSphere Application Server, see the*Administration Guide*.
- For information about security integration withWebSphere Application Server, see the*Administration Guide*.
- For information about configuring the WebSphere eXtreme Scale session manager to work with WebSphere Application Server, see the*Administration Guide*.

WebSphere Real Time

With support for WebSphere Real Time, the industry-leading real-time Java offering, WebSphere eXtreme Scale enables Extreme Transaction Processing (XTP) applications to have more consistent and predictable response times. See the information about Real Time Support in the *Administration Guide*.

Product name changes

Be aware that WebSphere eXtreme Scale has formerly been known by other names.

Product name changes

When referencing other documentation, marketing materials or presentations, keep in mind that eXtreme Scale has formerly been known by the following names.

- ObjectGrid
- WebSphere Extended Deployment Data Grid

Although the product itself is now known as WebSphere eXtreme Scale, the term ObjectGrid appears in the documentation and elsewhere because it is the name of the artifact that enables the data grid technology.

Directory conventions

The following directory conventions are used throughout the documentation to must reference special directories such as *wxs_install_root* and *wxs_home*. You access these directories during several different scenarios, including during installation and use of command-line tools.

wxs_install_root
The *wxs_install_root* directory is the root directory where WebSphere eXtreme Scale product files are installed. The *wxs_install_root* directory can be the directory in which the trial zip file is extracted or the directory in which the WebSphere eXtreme Scale product is installed.

- Example when extracting the trial:

  Example: `/opt/IBM/WebSphere/eXtremeScale`
- Example when WebSphere eXtreme Scale is installed to a stand-alone directory:

  Example: `/opt/IBM/eXtremeScale`
- Example when WebSphere eXtreme Scale is integrated with WebSphere Application Server:

  Example: `/opt/IBM/WebSphere/AppServer`

wxs_home
The *wxs_home* directory is the root directory of the WebSphere eXtreme Scale product libraries, samples and components. This is the same as the *wxs_install_root* directory when the trial is extracted. For stand-alone installations, the *wxs_home* directory is the `ObjectGrid` sub-directory within the *wxs_install_root* directory. For installations that are integrated with WebSphere Application Server, this directory is the `optionalLibraries/ObjectGrid` directory within the *wxs_install_root* directory.

- Example when extracting the trial:

  Example: `/opt/IBM/WebSphere/eXtremeScale`
- Example when WebSphere eXtreme Scale is installed to a stand-alone directory:

  Example: `/opt/IBM/eXtremeScale/ObjectGrid`
- Example when WebSphere eXtreme Scale is integrated with WebSphere Application Server:

  Example: `/opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid`

was_root
The *was_root* directory is the root directory of a WebSphere Application Server installation:

Example: `/opt/IBM/WebSphere/AppServer`

restservice_home
The *restservice_home* directory is the directory in which the WebSphere eXtreme Scale REST data service libraries and samples are located. This directory is named `restservice` and is a sub-directory under the *wxs_home* directory.

- Example for stand-alone deployments:

  Example: `/opt/IBM/WebSphere/eXtremeScale/ObjectGrid/restservice`
- Example for WebSphere Application Server integrated deployments:

  Example: `/opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid/restservice`

`tomcat_root`
The *`tomcat_root`* is the root directory of the Apache Tomcat installation.

Example: `/opt/tomcat5.5`

`wasce_root`
The *`wasce_root`* is the root directory of the WebSphere Application Server Community Edition installation.

Example:`/opt/IBM/WebSphere/AppServerCE`

`java_home`
The *`java_home`* is the root directory of a Java Runtime Environment (JRE) installation.

Example:`/opt/IBM/WebSphere/eXtremeScale/java`

`samples_home`
The *`samples_home`* is the directory in which you extract the sample files that are used for tutorials.

Example:`/wxs-samples/`

Free trial

To get started using WebSphere eXtreme Scale, download a free trial version. You can develop innovative, high-performance applications by extending the data caching concept using advanced features.

Trial download

You can download a free trial version of eXtreme Scale, from Download eXtreme Scale trial.

After downloading and unzipping the trial version of eXtreme Scale, navigate to the gettingstarted directory, and read GETTINGSTARTED_README.txt. This tutorial gets you started using eXtreme Scale, create a data grid on several servers, and run some simple applications to store and retrieve data in a grid. Before deploying eXtreme Scale in a production environment, there are several options to consider, including the number of servers to use, the amount of storage on each server, and synchronous or asynchronous replication.

Programming and Administration Guides

The *Product Overview* describes the fundamental concepts for understanding WebSphere eXtreme Scale. Two additional guides are available that expand on the concepts described in this guide.

Use the *Administration Guide* for configuration and general administrative tasks, and the *Programming Guide* for descriptions of the Java APIs for accessing and configuring the data grid.

Chapter 2. Caching overview

WebSphere eXtreme Scale can operate as an in-memory database processing space, which you can use to provide in-line caching for a database back-end or to serve as a side-cache. In-line caching uses eXtreme Scale as the primary means for interacting with the data. When eXtreme Scale is used as a side-cache, the back-end is used in conjunction with the data grid. This section describes various cache concepts and scenarios and discusses the available topologies for deploying a data grid.

Caching architecture: Maps, containers, clients, and catalogs

With WebSphere eXtreme Scale, your architecture can use local in-memory data caching or distributed client-server data caching.

WebSphere eXtreme Scale requires minimal additional infrastructure to operate. The infrastructure consists of scripts to install, start, and stop a Java Platform, Enterprise Edition application on a server. Cached data is stored in the eXtreme Scale server, and clients remotely connect to the server.

Distributed caches offer increased performance, availability and scalability and can be configured using dynamic topologies, in which servers are automatically balanced. You can also add additional servers without restarting your existing eXtreme Scale servers. You can create either simple deployments or large, terabyte-sized deployments in which thousands of servers are needed.

Maps

A map is a container for key-value pairs, which allows an application to store a value indexed by a key. Maps support indexes that can be added to index attributes on the key or value. These indexes are automatically used by the query runtime to determine the most efficient way to run a query.

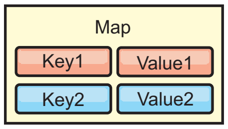


Figure 2. Map

A map set is a collection of maps with a common partitioning algorithm. The data within the maps are replicated based on the policy defined on the map set. A map set is only used for distributed topologies and is not needed for local topologies.

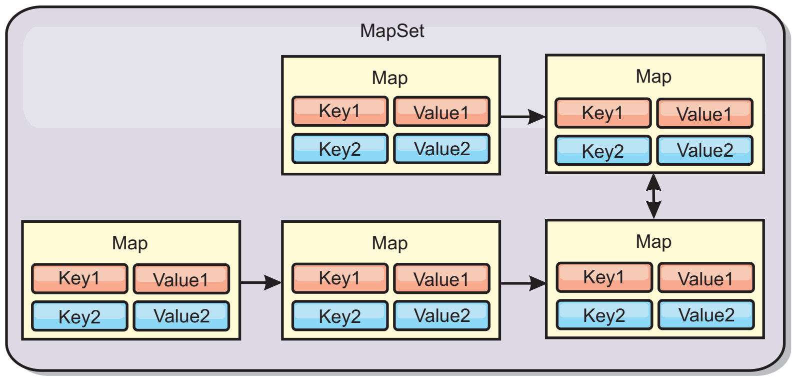


Figure 3. Map sets

A map set can have a schema associated with it. A schema is the metadata that describes the relationships between each map when using homogeneous Object types or entities.

WebSphere eXtreme Scale can store serializable Java objects in each of the maps using the ObjectMap API. A schema can be defined over the maps to identify the relationship between the objects in the maps where each map holds objects of a single type. Defining a schema for maps is required to query the contents of the map objects. WebSphere eXtreme Scale can have multiple map schemas defined.

See the ObjectMap API information in the *Programming Guide* for further details.

WebSphere eXtreme Scale can also store entities using the EntityManager API. Each entity is associated with a map. The schema for an entity map set is automatically discovered using either an entity descriptor XML file or annotated Java classes. Each entity has a set of key attributes and set of non-key attributes. An entity can also have relationships to other entities. WebSphere eXtreme Scale supports one to one, one to many, many to one and many to many relationships. Each entity is physically mapped to a single map in the map set. Entities allow applications to easily have complex object graphs that span multiple Maps. A distributed topology can have multiple entity schemas.

See the EntityManager API information in the *Programming Guide* for further details.

Container servers, partitions, and shards

The container server stores application data for the data grid. This data is generally broken into parts, which are called partitions, which are hosted across multiple container servers. Each container server in turn hosts a subset of the complete data. A JVM might host one or more container servers and each container server can host multiple shards.

Remember: Plan out the heap size for the container servers, which host all of your data. Configure the heap settings accordingly.

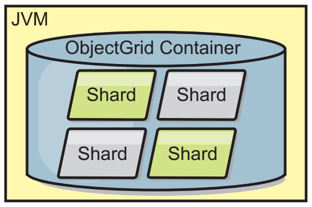


Figure 4. Container server

Partitions host a subset of the data in the grid. WebSphere eXtreme Scale automatically places multiple partitions in a single container server and spreads the partitions out as more container servers become available.

Important: Choose the number of partitions carefully before final deployment because the number of partitions cannot be changed dynamically. A hash mechanism is used to locate partitions in the network and eXtreme Scale cannot rehash the entire data set after it has been deployed. As a general rule, you can overestimate the number of partitions

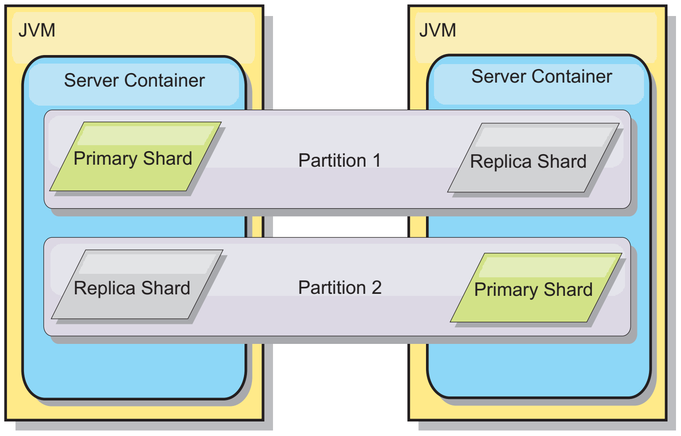


Figure 5. Partition

Shards are instances of partitions and have one of two roles: primary or replica. The primary shard and its replicas make up the physical manifestation of the partition. Every partition has several shards that each host all of the data contained in that partition. One shard is the primary, and the others are replicas, which are redundant copies of the data in the primary shard. A primary shard is the only partition instance that allows transactions to write to the cache. A replica shard is a "mirrored" instance of the partition. It receives updates synchronously or asynchronously from the primary shard. The replica shard only allows transactions to read from the cache. Replicas are never hosted in the same container server as the primary and are not normally hosted on the same machine as the primary.

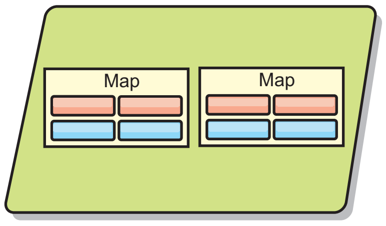


Figure 6. Shard

To increase the availability of the data, or increase persistence guarantees, replicate the data. However, replication adds cost to the transaction and trades performance in return for availability. With eXtreme Scale, you can control the cost as both synchronous and asynchronous replication is supported, as well as hybrid replication models using both synchronous and asynchronous replication modes. A synchronous replica shard receives updates as part of the transaction of the primary shard to guarantee data consistency. A synchronous replica can double the response time because the transaction has to commit on both the primary and the synchronous replica before the transaction is complete. An asynchronous replica shard receives updates after the transaction commits to limit impact on performance, but introduces the possibility of data loss as the asynchronous replica can be several transactions behind the primary.

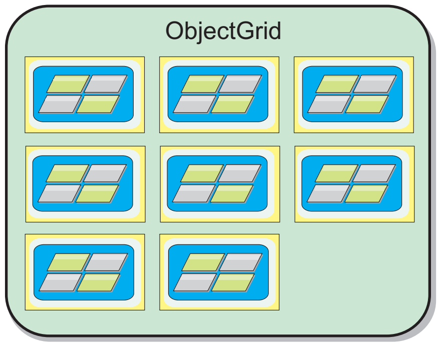


Figure 7. ObjectGrid

Clients

Clients connect to a catalog service, retrieve a description of the server topology, and communicate directly to each server as needed. When the server topology changes because new servers are added or existing servers have failed, the dynamic catalog service routes the client to the appropriate server that is hosting the data. Clients must examine the keys of application data to determine which partition to route the request. Clients can read data from multiple partitions in a single transaction. However, clients can update only a single partition in a transaction. After the client updates some entries, the client transaction must use that partition for updates.

The possible deployment combinations are included in the following list:

- A catalog service exists in its own grid of Java Virtual Machines. A single catalog service can be used to manage multiple eXtreme Scale clients or servers.
- A container can be started in a JVM by itself or can be loaded into an arbitrary JVM with other containers for different ObjectGrid instances.

- A client can exist in any JVM and communicate with one or more ObjectGrid instances. A client can also exist in the same JVM as a container.

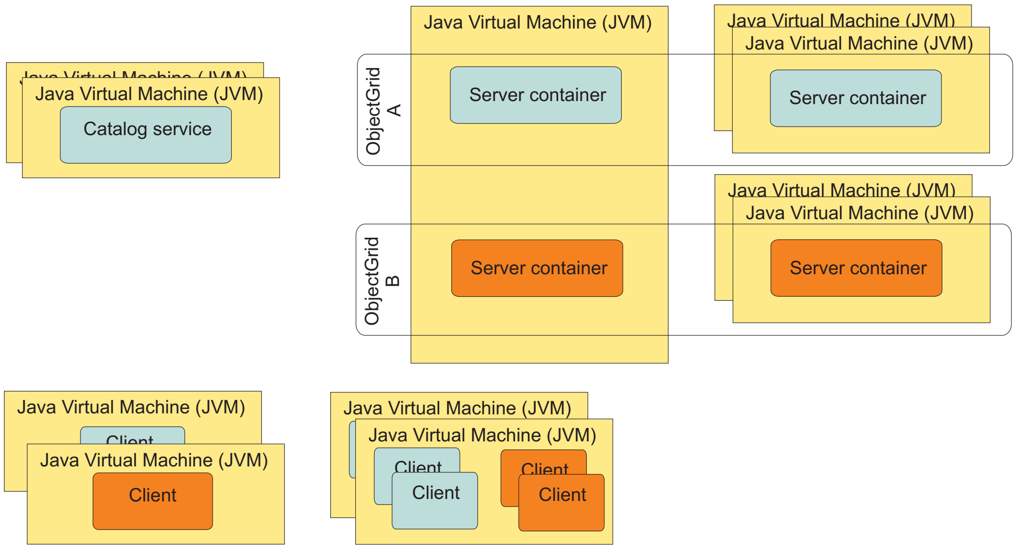

Java Virtual Machine (JVM)
Java Virtual Machine (JVM)
Catalog service
Java Virtual Machine (JVM)
ObjectGrid A
Server container
Java Virtual Machine (JVM)
Java Virtual Machine (JVM)
Server container
ObjectGrid B
Server container
Java Virtual Machine (JVM)
Java Virtual Machine (JVM)
Server container
Java Virtual Machine (JVM)
Client
Java Virtual Machine (JVM)
Client
Java Virtual Machine (JVM)
Java Virtual Machine (JVM)
Client
Client
Client
Client

Figure 8. Possible topologies

Catalog service

The catalog service hosts logic that should be idle during a steady state and has little influence on scalability. The catalog service is built to service hundreds of containers becoming available simultaneously and runs services to manage the containers.

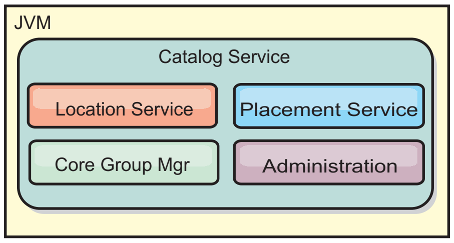


Figure 9. Catalog service

The catalog responsibilities consist of the following services:

Location service
The location service provides locality for clients that are looking for containers hosting applications and for containers that are looking to register hosted applications with the placement service. The location service runs in all of the grid members to scale out this function.

Placement service
The placement service is the central nervous system for the grid and is responsible for allocating individual shards to their host container. The placement service runs as a One of N elected service in the cluster. Because the One of N policy is used, there is always exactly one instance of the placement service running. If that instance should stop, another process

takes over. All states of the catalog service are replicated across all servers hosting the catalog service for redundancy.

Core group manager
The core group manager manages peer grouping for health monitoring, organizes containers into small groups of servers, and automatically federates the groups of servers. When a container first contacts the catalog service, the container waits to be assigned to either a new or an existing group of several Java virtual machines (JVM). Each group of Java virtual machines monitors the availability of each of its members through heartbeating. One of the group members relays availability information to the catalog service to allow for reacting to failures by reallocation and route forwarding.

Administration
The four stages of administering your WebSphere eXtreme Scale environment are planning, deploying, managing, and monitoring.

For availability, configure a catalog service domain. A catalog service domain consists of multiple Java virtual machines, including a master JVM and a number of backup Java virtual machines.

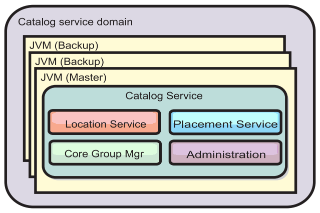


Figure 10. Catalog service domain

Caching topology: In-memory and distributed caching

With WebSphere eXtreme Scale, your architecture can use local in-memory data caching or distributed client-server data caching.

WebSphere eXtreme Scale requires minimal additional infrastructure to operate. The infrastructure consists of scripts to install, start, and stop a Java Platform, Enterprise Edition application on a server. Cached data is stored in the eXtreme Scale server, and clients remotely connect to the server.

In-memory environments

When you deploy in a local, in-memory environment, WebSphere eXtreme Scale runs within a single Java virtual machine and is not replicated. To configure a local environment you can use an ObjectGrid XML file or the ObjectGrid APIs.

Distributed environments

When you deploy in a distributed environment, WebSphere eXtreme Scale runs across a set of Java virtual machines, increasing the performance, availability and scalability. With this configuration, you can use data replication and partitioning. You can also add additional servers without restarting your existing eXtreme Scale servers. As with a local environment, an ObjectGrid XML file, or an equivalent programmatic configuration, is needed in a distributed environment. You must also provide a deployment policy XML file with configuration details

You can create either simple deployments or large, terabyte-sized deployments in which thousands of servers are needed.

Local in-memory cache

In the simplest case, WebSphere eXtreme Scale can be used as a local (non-distributed) in-memory data grid cache. The local case can especially benefit high-concurrency applications where multiple threads need to access and modify transient data. The data kept in a local data grid can be indexed and retrieved using queries. Queries help you to work with large in memory data sets. The support provided with the Java virtual machine (JVM), although it is ready to use, has a limited data structure.

The local in-memory cache topology for WebSphere eXtreme Scale is used to provide consistent, transactional access to temporary data within a single Java virtual machine.

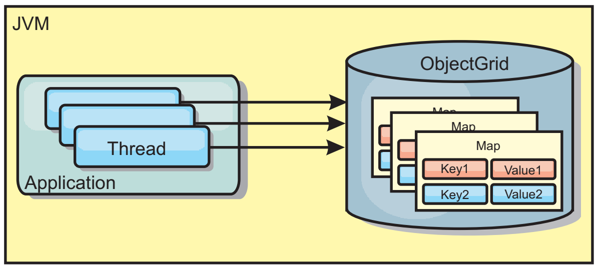


Figure 11. Local in-memory cache scenario

Advantages

- Simple setup: An ObjectGrid can be created programmatically or declaratively with the ObjectGrid deployment descriptor XML file or with other frameworks such as Spring.
- Fast: Each BackingMap can be independently tuned for optimal memory utilization and concurrency.
- Ideal for single-Java virtual machine topologies with small dataset or for caching frequently accessed data.
- Transactional. BackingMap updates can be grouped into a single unit of work and can be integrated as a last participant in 2-phase transactions such as Java Transaction Architecture (JTA) transactions.

Disadvantages

- Not fault tolerant.
- The data is not replicated. In-memory caches are best for read-only reference data.
- Not scalable. The amount of memory required by the database might overwhelm the Java virtual machine.
- Problems occur when adding Java virtual machines:
  - Data cannot easily be partitioned
  - Must manually replicate state between Java virtual machines or each cache instance could have different versions of the same data.
  - Invalidation is expensive.
  - Each cache must be warmed up independently. The warm-up is the period of loading a set of data so that the cache gets populated with valid data.

When to use

The local, in-memory cache deployment topology should only be used when the amount of data to be cached is small (can fit into a single Java virtual machine) and is relatively stable. Stale data must be tolerated with this approach. Using evictors to keep most frequently or recently used data in the cache can help keep the cache size low and increase relevance of the data.

Peer-replicated local cache

You must ensure the cache is synchronized if multiple processes with independent cache instances exist. To ensure that the cache instances are synchronized, enable a peer-replicated cache with Java Message Service (JMS).

WebSphere eXtreme Scale includes two plug-ins that automatically propagate transaction changes between peer ObjectGrid instances. The JMSObjectGridEventListener plug-in automatically propagates eXtreme Scale changes using JMS.

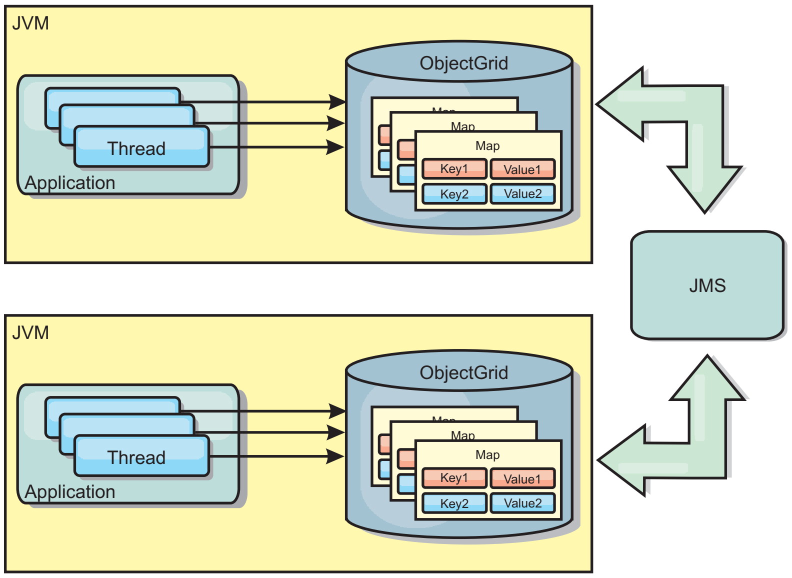


Figure 12. Peer-replicated cache with changes that are propagated with JMS

If you are running a WebSphere Application Server environment, the TranPropListener plug-in is also available. The TranPropListener plug-in uses the high availability (HA) manager to propagate the changes to each peer cache instance.

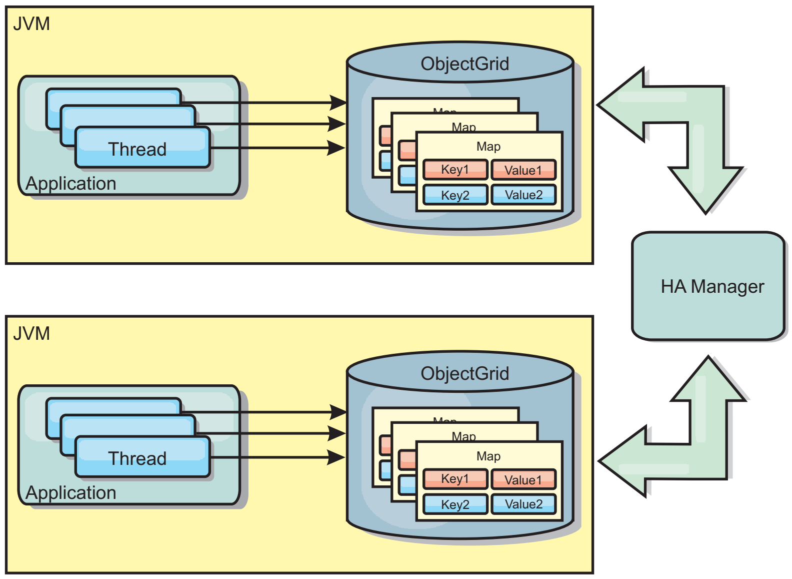


Figure 13. Peer-replicated cache with changes that are propagated with the high availability manager

Advantages

- The data is more valid because the data is updated more often.
- With the TranPropListener plug-in, like the local environment, the eXtreme Scale can be created programmatically or declaratively with the eXtreme Scale deployment descriptor XML file or with other frameworks such as Spring. Integration with the high availability manager is done automatically.
- Each BackingMap can be independently tuned for optimal memory utilization and concurrency.
- BackingMap updates can be grouped into a single unit of work and can be integrated as a last participant in 2-phase transactions such as Java Transaction Architecture (JTA) transactions.
- Ideal for few-JVM topologies with a reasonably small dataset or for caching frequently accessed data.
- Changes to the eXtreme Scale are replicated to all peer eXtreme Scale instances. The changes are consistent as long as a durable subscription is used.

Disadvantages

- Configuration and maintenance for the JMSObjectGridEventListener can be complex. eXtreme Scale can be created programmatically or declaratively with the eXtreme Scale deployment descriptor XML file or with other frameworks such as Spring.
- Not scalable: The amount of memory required by the database may overwhelm the JVM.
- Functions improperly when adding Java virtual machines:
  - Data cannot easily be partitioned
  - Invalidation is expensive.
  - Each cache must be warmed-up independently

When to use

Use deployment topology only when the amount of data to be cached is small, can fit into a single JVM, and is relatively stable.

Distributed cache

WebSphere eXtreme Scale is most often used as a shared cache, to provide transactional access to data to multiple components where a traditional database would otherwise be used. The shared cache eliminates the need configure a database.

Coherency of the cache

The cache is coherent because all of the clients see the same data in the cache. Each piece of data is stored on exactly one server in the cache, preventing wasteful copies of records that could potentially contain different versions of the data. A coherent cache can also hold more data as more servers are added to the data grid, and scales linearly as the grid grows in size. Because clients access data from this data grid with remote procedural calls, it can also be known as a remote cache, or far cache. Through data partitioning, each process holds a unique subset of the total data set. Larger data grids can both hold more data and service more requests for that data. Coherency also eliminates the need to push invalidation data around the data grid because no stale data exists. The coherent cache only holds the latest copy of each piece of data.

If you are running a WebSphere Application Server environment, the TranPropListener plug-in is also available. The TranPropListener plug-in uses the high availability component (HA Manager) of WebSphere Application Server to propagate the changes to each peer ObjectGrid cache instance.

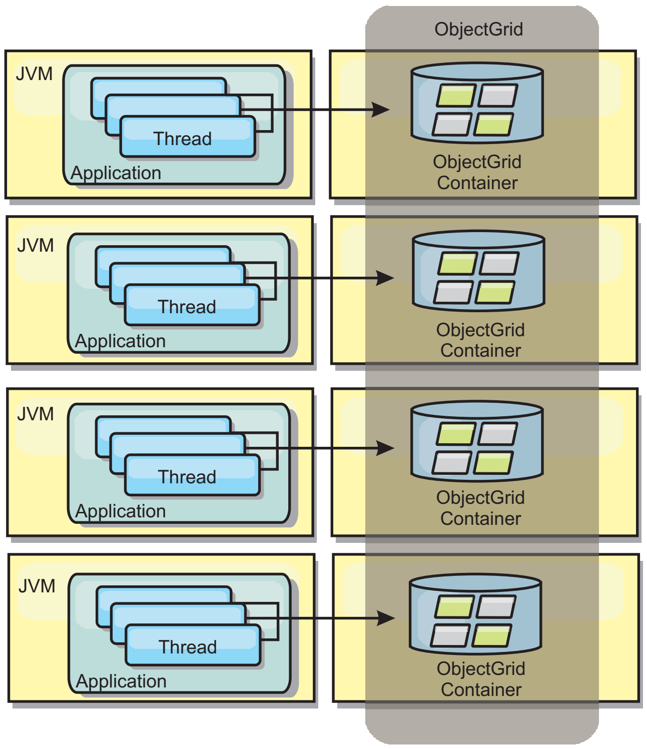


Figure 14. Distributed cache

Near cache

Clients can optionally have a local, in-line cache when eXtreme Scale is used in a distributed topology. This optional cache is called a near cache, an independent

ObjectGrid on each client, serving as a cache for the remote, server-side cache. The near cache is enabled by default when locking is configured as optimistic or none and cannot be used when configured as pessimistic.

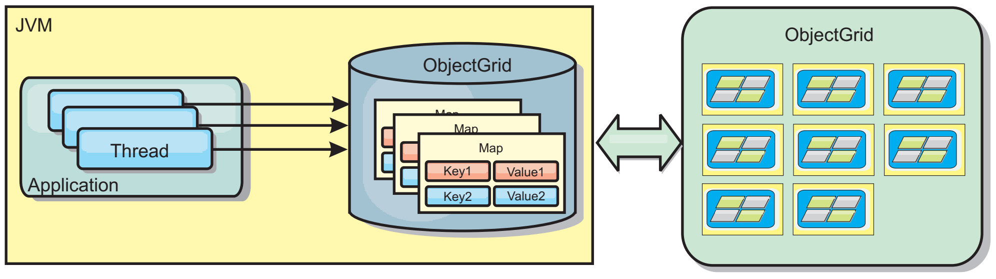


Figure 15. Near cache

A near cache is very fast because it provides in-memory access to a subset of the entire cached data set that is stored remotely in the eXtreme Scale servers. The near cache is not partitioned and contains data from any of the remote eXtreme Scale partitions.WebSphere eXtreme Scale can have up to three cache tiers as follows.

1. The transaction tier cache contains all changes for a single transaction. The transaction cache contains a working copy of the data until the transaction is committed. When a client transaction requests data from an ObjectMap, the transaction is checked first
2. The near cache in the client tier contains a subset of the data from the server tier. When the transaction tier does not have the data, the data is fetched from the client tier, if available and inserted into the transaction cache
3. The data grid in the server tier contains the majority of the data and is shared among all clients. The server tier can be partitioned, which allows a large amount of data to be cached. When the client near cache does not have the data, it is fetched from the server tier and inserted into the client cache. The server tier can also have a Loader plug-in. When the grid does not have the requested data, the Loader is invoked and the resulting data is inserted from the backend data store into the grid.

To disable the near cache, set the numberOfBuckets attribute to 0 in the client override eXtreme Scale descriptor configuration. See the topic on map entry locking for details on eXtreme Scale lock strategies. The near cache can also be configured to have a separate eviction policy and different plug-ins using a client override eXtreme Scale descriptor configuration.

Advantage

- Fast response time because all access to the data is local. Looking for the data in the near cache first saves a trip to the grid of servers, thus making even the remote data locally accessible.

Disadvantages

- Increases duration of stale data because the near cache at each tier may be out of synch with the current data in the grid.
- Relies upon an evictor to invalidate data to avoid running out of memory.

When to use

Use when response time is important and stale data can be tolerated.

Embedded cache

WebSphere eXtreme Scale grids can run within existing processes as embedded eXtreme Scale servers or you can manage them as external processes.

Embedded grids are useful when you are running in an application server, such as WebSphere Application Server. You can start eXtreme Scale servers that are not embedded by using command line scripts and run in a Java process.

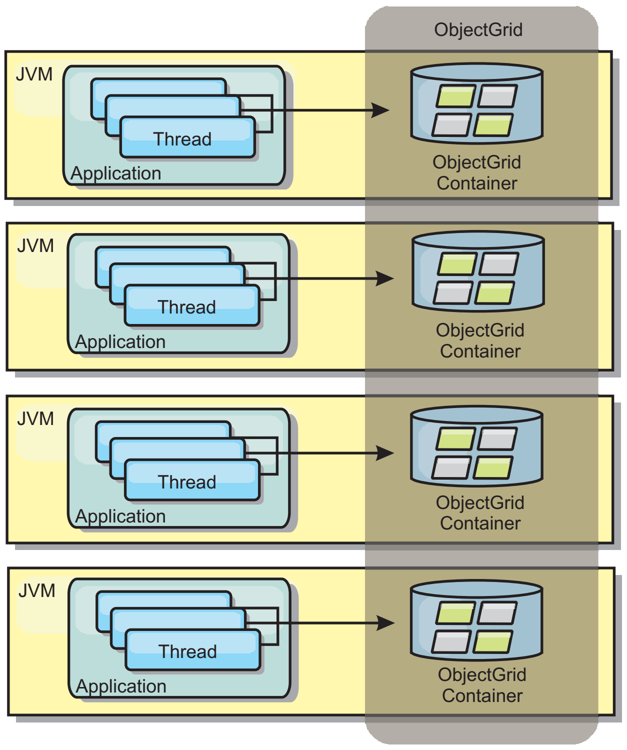


Figure 16. Embedded cache

Advantages

- Simplified administration since there are less processes to manage.
- Simplified application deployment since the grid is using the client application classloader.
- Supports partitioning and high availability.

Disadvantages

- Increased the memory footprint in client process since all of the data is collocated in the process.
- Increase CPU utilization for servicing client requests.
- More difficult to handle application upgrades since clients are using the same application Java archive files as the servers.
- Less flexible. Scaling of clients and grid servers cannot increase at the same rate. When servers are externally defined, you can have more flexibility in managing the number of processes.

When to use

Use embedded grids when there is plenty of memory free in the client process for grid data and potential failover data.

For more information, see the topic on enabling the client invalidation mechanism in the *Administration Guide*.

Multi-master data grid replication topologies

Using the multi-master asynchronous replication feature, two or more data grids can become exact mirrors of one other. This mirroring is accomplished using asynchronous replication among links connecting the data grids together. Each data grid is hosted in an independent catalog service domain, with its own catalog service, container servers, and a unique name. With the multi-master asynchronous replication feature, you can use links to interconnect a collection of these catalog service domains. Then, you can synchronize the catalog service domains with replication over the links. You can construct almost any topology because you choose how to define links among catalog service domains.

7.1+ Multi-master data grid replication is a significant new feature in Version 7.1. The feature is also called AP (availability and partitioning) replication in the context of the CAP theorem. The CAP theorem states that a distributed computer system cannot support more than two of the following three properties: consistency, availability, and partition tolerance.

See “Initial considerations for multi-master topologies” for map sets that are not replicated.

A replication data grid infrastructure is a connected graph of catalog service domains with bidirectional links among them. You can use a link between two catalog service domains to track data changes. For more information about how to set up communication between catalog service domains for multi-master replication, see “Available topologies for multi-master replication” on page 25.

Also, depending on the requirements of your environment, you can optimize the topology design for multi-master replication by taking several factors into consideration: arbitration, linking, and performance. Read more at “Topology considerations for multi-master replication” on page 28.

Initial considerations for multi-master topologies:

Consider the following issues when you are deciding whether and how to use multi-master replication topologies.

- **Configuring class loaders with multiple catalog service domains**

  Domains must have access to all classes that are used as keys and values. Any dependencies must be reflected in all class paths for data grid container JVMs for all domains. If a CollisionArbiter plug-in retrieves the value for a cache entry, then the classes for the values must be present for the domain that is starting the arbiter.
- **Avoid loaders**

  Loaders can be used to interface changes between a data grid and a database. It is unlikely that all data grids or domains in a topology are collocated geographically with the same database. WAN latency and other factors might render this use case undesirable.

  Grid preloading also requires careful design. Usually, when a data grid is restarted, it is preloaded again. Preloading is not necessary or required when using multi-master replication. As soon as a catalog service domain is online, it automatically reloads itself with the contents of the domains to which it is linked. As a result, you are not required to initiate a manual preload for a data grid that is a domain in a multi-master replication topology.

Loaders usually obey insert and update rules. With multi-master replication, inserts must be treated as merges. When the data is being pulled remotely after a domain restart, existing data will be merged into the local domain. Because the data might already have been in the local database, a typical insert fails with a duplicate key exception in the database. Use merge semantics instead.

WebSphere eXtreme Scale can be configured to do a shard-based preload with the preload methods on Loader plug-ins. But you should avoid this technique in a multi-master replication topology. Instead, use a client-based preload when the topology is first started. The multi-master topology refreshes any restarted domains with a current copy of what is stored in other domains in the topology. After domains have been started, the multi-master topology keeps domains synchronized.

- **EntityManager is not supported**

  A map set containing an entity map is not replicated across catalog service domains.

- **Byte array maps are not supported in releases before Version 7.1.0.2**

  A map set containing a map that is configured with COPY_TO_BYTES copy mode is not replicated across catalog service domains.

  7.1.0.2+ In Version 7.1.0.2 or later, maps that are configured with COPY_TO_BYTES copy mode can replicate across catalog service domains. To enable this function, you must upgrade your entire configuration to Version 7.1.0.2 or later. All catalog servers, clients, and container servers, including container servers that are running only replica shards, in all domains must be upgraded. You cannot have COPY_TO_BYTES copy mode enabled on a catalog service domain that contains any servers that are running a version before Version 7.1.0.2. To upgrade your catalog service domains to support COPY_TO_BYTES copy mode, use the following steps:

  1. Use the **xsadmin -dismissLink** command to remove the multi-master link between your catalog service domains. See the information about configuring multi-master replication topologies in the *Administration Guide*for more information.
  2. Shut down the data grid. You can use the **xsadmin -teardown** command to stop a group of catalog and container servers. See the information about stopping servers in the *Administration Guide* for more information.
  3. Upgrade servers and clients in each domain to Version 7.1.0.2 or later. See the information about updating eXtreme Scale servers in the *Administration Guide* for more information.
  4. Update your configuration to use the COPY_TO_BYTES copy mode. See the information about byte array maps in the *Programming Guide* for more information about editing the byte array configuration.
  5. Restart the data grid. See the information about starting servers in the *Administration Guide* for more information.
  6. Use the **xsadmin -establishLink** command to reconnect the catalog service domains. See the information about configuring multi-master replication topologies in the *Administration Guide* for more information.

  After all of your catalog service domains are upgraded, you cannot start servers in any domains that are at a level that is lower than Version 7.1.0.2.

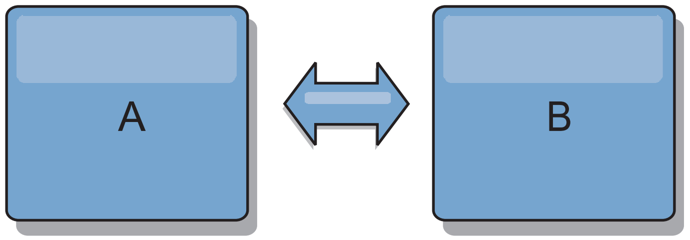
- **Write-behind is not supported**

A map set containing a map that is configured with write-behind support is not replicated across catalog service domains.

Available topologies for multi-master replication:

You have several different options when choosing the topology for your deployment that incorporates multi-master replication.

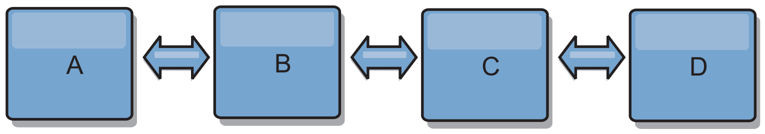
A replication data grid infrastructure is a connected graph of catalog service domains with bidirectional links among them. With a link, two catalog service domains can communicate data changes. For example, the simplest topology is a pair of catalog service domains with a single link between them. The catalog service domains are named alphabetically: A, B, C, and so on, from the left. A link can cross a wide area network (WAN), spanning large distances. Even if the link is interrupted, you can still change data in either catalog service domain. The topology reconciles changes when the link reconnects the catalog service domains. Links automatically try to reconnect if the network connection is interrupted.



After you set up the links, then eXtreme Scale first tries to make every catalog service domain identical. Then, eXtreme Scale tries to maintain the identical conditions as changes occur in any catalog service domain. The goal is for each catalog service domain to be an exact mirror of every other catalog service domain connected by the links. The replication links between the catalog service domains help ensure that any changes made in one domain are copied to the other domains.

Line topologies

Although it is such a simple deployment, a line topology demonstrates some qualities of the links. First, it is not necessary for a catalog service domain to be connected directly to every other catalog service domain to receive changes. Domain B pulls changes from Domain A. Domain C receives changes from Domain A through Domain B, which connects Domains A and C. Similarly, Domain D receives changes from the other domains through Domain C. This ability spreads the load of distributing changes away from the source of the changes.



Notice that if Domain C fails, the following would occur:

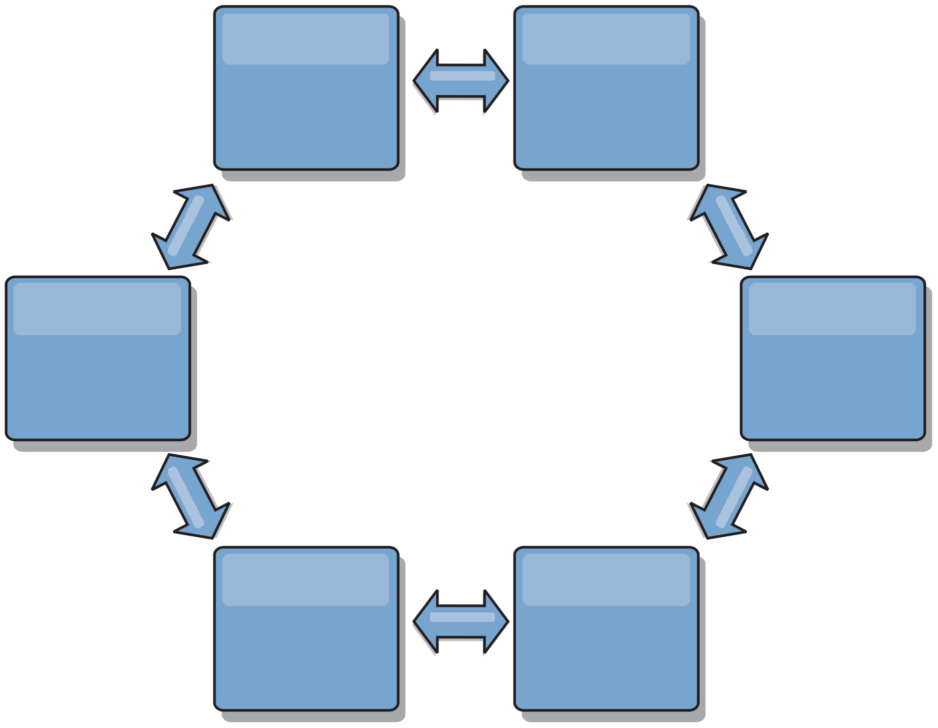
1. Domain D would be orphaned until Domain C was restarted
2. Domain C would synchronize itself with Domain B, which is a copy of Domain A

3. Domain D would use Domain C to synchronize itself with changes on Domains A and B. These changes initially occurred while Domain D was orphaned (while Domain C was down).

Ultimately, Domains A, B, C, and D would all become identical to one other again.

Ring topologies

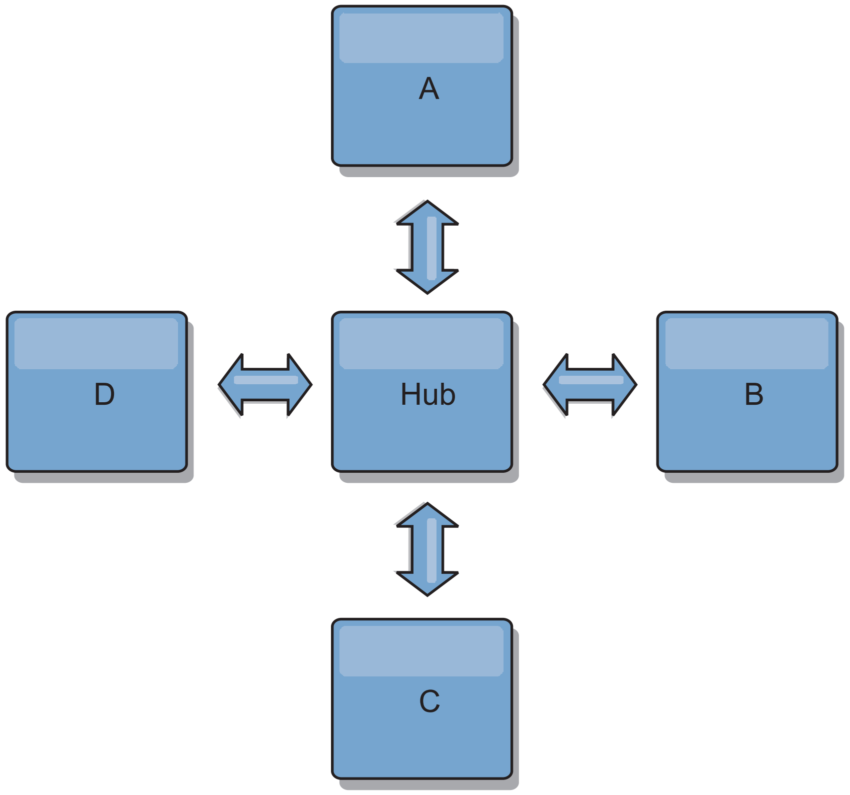
Another option you have with multi-master replication is a ring topology, which is more resilient than the topologies described in the previous sections. A catalog service domain or a single link can fail. Still, the surviving catalog service domains can obtain changes by traveling around the ring, away from the failure. Each catalog service domain has two links to the other catalog service domains. And each catalog service domain has at most two links, no matter how large the ring topology. Changes from a particular domain might travel through several domains before all of them mirror each other. Going through several domains causes potentially high latency, similar to the processes for a line topology.



You can also deploy a more sophisticated ring topology, with a root catalog service domain at the center of the ring. The root catalog service domain functions as the central point of reconciliation. The other catalog service domains act as remote points of reconciliation for changes occurring in the root catalog service domain. The root catalog service domain can arbitrate changes among the catalog service domains. If a ring topology contains more than one ring around a root catalog service domain, the domain can only arbitrate changes among the innermost ring. However, the results of the arbitration spread throughout the catalog service domains in the other rings.

Hub-and-spoke topologies

With a hub-and-spoke topology, changes travel through a hub catalog service domain. Because the hub is the only intermediate catalog service domain that is specified, hub-and-spoke topologies have lower latency. The hub domain is connected to every spoke domain through a link. The hub distributes changes among the catalog service domains. The hub acts as a point of reconciliation for collisions. In an environment with a high update rate, the hub might require run on more hardware than the spokes to remain synchronized. WebSphere eXtreme Scale is designed to scale linearly, meaning you can make the hub larger, as needed, without difficulty. However, if the hub fails, then changes are not distributed until the hub restarts. Any changes on the spoke catalog service domains will be distributed after the hub is reconnected.



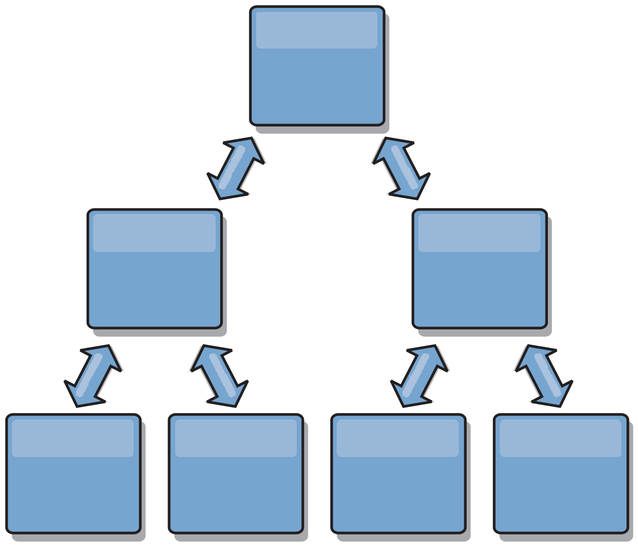
You can also use a strategy with fully replicated clients, a topology variation which uses a pair of eXtreme Scale servers running as a hub. Every client creates a self-contained single container data grid with a catalog in the client JVM. A client uses its data grid to connect to the hub catalog. This connection causes the client to synchronize with the hub as soon as the client obtains a connection to the hub.

Any changes made by the client are local to the client, and are replicated asynchronously to the hub. The hub acts as an arbitration domain, distributing changes to all connected clients. The fully replicated clients topology provides a reliable L2 cache for an object relational mapper, such as OpenJPA. Changes are distributed quickly among client JVMs through the hub. If the cache size can be contained within the available heap space, the topology is a reliable architecture for this style of L2.

Use multiple partitions to scale the hub domain on multiple JVMs, if necessary. Because all of the data still must fit in a single client JVM, multiple partitions increase the capacity of the hub to distribute and arbitrate changes. However, having multiple partitions does not change the capacity of a single domain.

Tree topologies

You can also use an acyclic directed tree. An acyclic tree has no cycles or loops, and a directed setup limits links to existing only between parents and children. You can use the tree topology when you have many catalog service domains such that the ring topology would overwork the hub. You can also use a tree if you require being able to add child catalog service domains without updating the root catalog service domain.



A tree topology can still have a central point of reconciliation in the root catalog service domain. The second level can still function as a remote point of reconciliation for changes occurring in the catalog service domain beneath them. The root catalog service domain can arbitrate changes between the catalog service domains on the second level only. You can also use N-ary trees, each of which have N children at each level. Each catalog service domain connects out to *n* links.

Topology considerations for multi-master replication:

When implementing multi-master replication, you must consider aspects in your design such as: arbitration, linking, and performance.

Linking considerations in topology design

Ideally, a topology includes the minimum number of links while optimizing trade-offs among change latency, fault tolerance, and performance characteristics.

- **Change latency**

  Change latency is determined by the number of intermediate catalog service domains a change must go through before arriving at a specific catalog service domain.

  A topology has the best change latency when it eliminates intermediate catalog service domains by linking every catalog service domain to every other catalog service domain. However, a catalog service domain must perform replication work in proportion to its number of links. For large topologies, the sheer number of links to be defined can cause an administrative burden.

  The speed at which a change is copied to other catalog service domains depends on additional factors, such as:

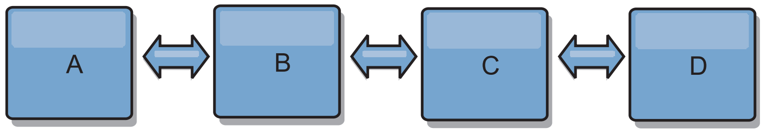
  – Processor and network bandwidth on the source catalog service domain

  - The number of intermediate catalog service domains and links between the source and target catalog service domain
  - The processor and network resources available to the source, target, and intermediate catalog service domains
- **Fault tolerance**

  Fault tolerance is determined by how many paths exist between two catalog service domains for change replication.

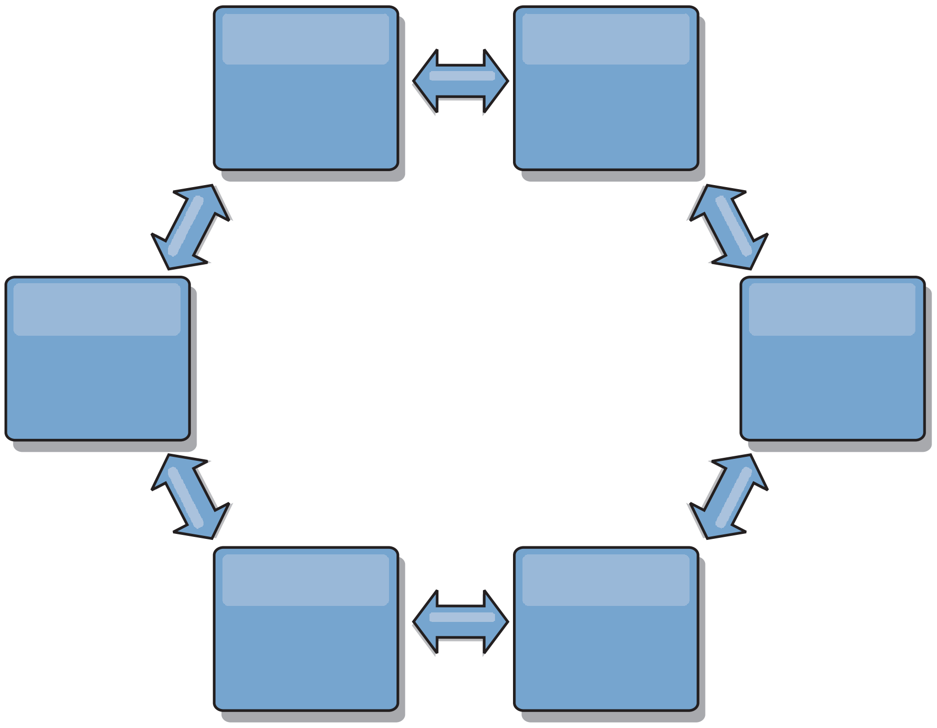
  If you have only one link between a given pair of catalog service domains, a link failure disallows propagation of changes. Similarly, changes are not propagated between catalog service domains if any of the intermediate domains experiences link failure. Your topology could have a single link from one catalog service domain to another such that the link passes through intermediate domains. If so, then changes are not propagated if any of the intermediate catalog service domains is down.

  Consider the line topology with four catalog service domains A, B, C, and D:



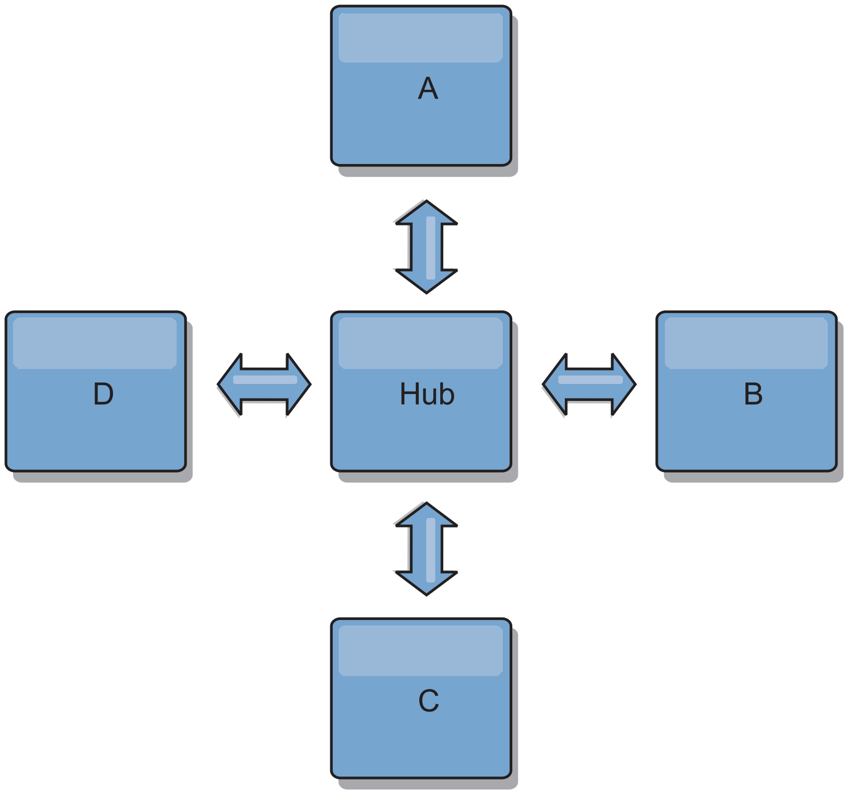
  If any of these conditions hold, Domain D does not see any changes from A:
  - Domain A is up and B is down
  - Domains A and B are up and C is down
  - The link between A and B is down
  - The link between B and C is down
  - The link between C and D is down

  In contrast, with a ring topology, each catalog service domain can receive changes from either direction.



  For example, if a given catalog service in your ring topology is down, then the two adjacent domains can still pull changes directly from each other.

All changes are propagated through the hub. Thus, as opposed to the line and ring topologies, the hub-and-spoke design is susceptible to breakdown if the hub fails.



A single catalog service domain is resilient to a certain amount of service loss. However, larger failures such as wide network outages or loss of links between physical data centers can disrupt any of your catalog service domains.

- **Linking and performance**

  The number of links defined on a catalog service domain affects performance. More links use more resources and replication performance can drop as a result. The ability to retrieve changes for a domain A through other domains effectively off-loads domain A from replicating its transactions everywhere. The change distribution load on a domain is limited by the number of links it uses, not how many domains are in the topology. This load property provides scalability, so the domains in the topology can share the burden of change distribution.

  A catalog service domain can retrieve changes indirectly through other catalog service domains. Consider a line topology with five catalog service domains.

  ```
  A <=> B <=> C <=> D <=> E
  ```

  – A pulls changes from B, C, D, and E through B
  – B pulls changes from A and C directly, and changes from D and E through C
  – C pulls changes from B and D directly, and changes from A through B and E through D
  – D pulls changes from C and E directly, and changes from A and B through C
  – E pulls changes from D directly, and changes from A, B, and C through D

  The distribution load on catalog service domains A and E is lowest, because they each have a link only to a single catalog service domain. Domains B, C, and D each have a link to two domains. Thus, the distribution load on domains B, C, and D is double the load on domains A and E. The workload depends on the number of links in each domain, not on the overall number of domains in the topology. Thus, the described distribution of loads would remain constant, even if the line contained 1000 domains.

Arbitration considerations in topology design

Change collisions might occur if the same records can be changed simultaneously in two places. Set up each catalog service domain to have about the same amount of processor, memory, network resources. You might observe that catalog service domains performing change collision handling (arbitration) use more resources than other catalog service domains. Collisions are detected automatically. They are handled with one of two mechanisms:

- **Default collision arbiter** The default protocol is to use the changes from the lexically lowest named catalog service domain. For example, if catalog service domain A and B generate a conflict for a record, then the change from catalog service domain B is ignored. Catalog service domain A keeps its version and the record in catalog service domain B is changed to match the record from catalog service domain A. This behavior applies as well for applications where users or sessions are normally bound or have affinity with one of the data grids.
- **Custom collision arbiter** Applications can provide a custom arbiter. When a catalog service domain detects a collision, it starts the arbiter. For information about developing a useful custom arbiter, see Developing custom arbiters for multi-master replication.

For topologies in which collisions are possible, consider implementing a hub-and-spoke topology or a tree topology. These two topologies are conducive to avoiding constant collisions, which can happen in the following scenarios:

1. Multiple catalog service domains experience a collision
2. Each catalog service domain handles the collision locally, producing revisions
3. The revisions collide, resulting in revisions of revisions

To avoid collisions, choose a specific catalog service domain, called an *arbitration catalog service domain* as the collision arbiter for a subset of catalog service domains. For example, a hub-and-spoke topology might use the hub as the collision handler. The spoke collision handler ignores any collisions that are detected by the spoke catalog service domains. The hub catalog service domain creates revisions, preventing unexpected collision revisions. The catalog service domain that is assigned to handle collisions must link to all of the domains for which it is responsible for handling collisions. In a tree topology, any internal parent domains handle collisions for their immediate children. In contrast, if you use a ring topology, you cannot designate one catalog service domain in the ring as the arbiter.

The following table summarizes the arbitration approaches that are most compatible with various topologies.

Table 2. Arbitration approaches. This table states whether application arbitration is compatible with various technologies.

Topology	Application ration?	Notes
A line of two catalog service domains	Yes	Choose one catalog service domain as the arbiter.

Table 2. Arbitration approaches (continued). This table states whether application arbitration is compatible with various technologies.

Topology	Application ration?	Notes
A line of three catalog service domains	Yes	The middle catalog service domain must be the arbiter. Think of the middle catalog service domain as the hub in a simple hub-and-spoke topology.
A line of more than three catalog service domains	No	Application arbitration is not supported.
A hub with N spokes	Yes	Hub with links to all spokes must be the arbitration catalog service domain.
A ring of N catalog service domains	No	Application arbitration is not supported.
An acyclic, directed tree (N-ary tree)	Yes	All root nodes must rate their direct descendants only.

Multi-master replication performance considerations

Take the following limitations into account when using multi-master replication topologies:

- **Change distribution tuning** (Discussed in previous section, "Linking and performance.")
- **Replication link performance** WebSphere eXtreme Scale creates a single TCP/IP socket between any pair of JVMs. All traffic between the JVMs occurs through the single socket, including traffic from multi-master replication. Catalog service domains are hosted on at least *n* container JVMs, providing at least *n* TCP links to peer catalog service domains. Thus, the catalog service domains with larger numbers of containers have higher replication performance levels. More containers require more processor and network resources.
- **TCP sliding window tuning and RFC 1323** RFC 1323 support on both ends of a link yields more data for a round trip. This support results in higher throughput, expanding the capacity of the window by a factor of about 16,000.

  Recall that TCP sockets use a sliding window mechanism to control the flow of bulk data. This mechanism typically limits the socket to 64 KB for a round-trip interval. If the round-trip interval is 100 ms, then the bandwidth is limited to 640 KB/second without additional tuning. Fully using the bandwidth available on a link might require tuning that is specific to an operating system. Most operating systems include tuning parameters, including RFC 1323 options, to enhance throughput over high-latency links.

  Several factors can affect replication performance:
  - The speed at which eXtreme Scale retrieves changes.
  - The speed at which eXtreme Scale can service retrieve replication requests.
  - The sliding window capacity.
  - With network buffer tuning on both sides of a link, eXtreme Scale retrieves changes over the socket efficiently.
- **Object Serialization** All data must be serializable. If a catalog service domain is not using COPY_TO_BYTES, then the catalog service domain must use Java serialization or ObjectTransformers to optimize serialization performance.

- **Compression** WebSphere eXtreme Scale compresses all data sent between catalog service domains by default. Disabling compression is not currently available.
- **Memory tuning** The memory usage for a multi-master replication topology is largely independent of the number of catalog service domains in the topology.

  Multi-master replication adds a fixed overhead per Map entry to handle versioning. Each container also tracks a fixed amount of data for each catalog service domain in the topology. A topology with two catalog service domains uses approximately the same memory as a topology with 50 catalog service domains. WebSphere eXtreme Scale does not use replay logs or similar queues in its implementation. Thus, there is no recovery structure ready in the case that a replication link is unavailable for a substantial period and later restarts.

Database integration: Write-behind, in-line, and side caching

WebSphere eXtreme Scale is used to front a traditional database and eliminate read activity that is normally pushed to the database. A coherent cache can be used with an application directly or indirectly using an object relational mapper. The coherent cache can then offload the database or backend from reads. In a slightly more complex scenario, such as transactional access to a data set where only some of the data requires traditional persistence guarantees, filtering can be used to offload even write transactions.

You can configure eXtreme Scale to function as a highly flexible in-memory database processing space. However, eXtreme Scale is not an object relational mapper (ORM). It does not know where the data in eXtreme Scale came from. An application or an ORM can place data in an eXtreme Scale server. It is the responsibility of the source of the data to make sure that it stays consistent with the database where data originated. This means eXtreme Scale cannot invalidate data that is pulled from a database automatically. The application or mapper must provide this function and manage the data stored in eXtreme Scale.

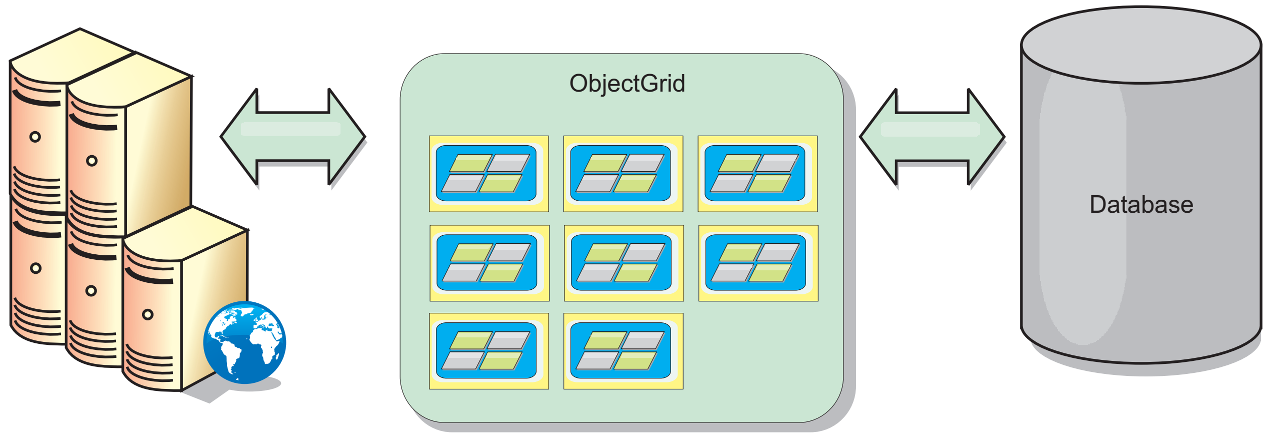


Figure 17. ObjectGrid as a database buffer

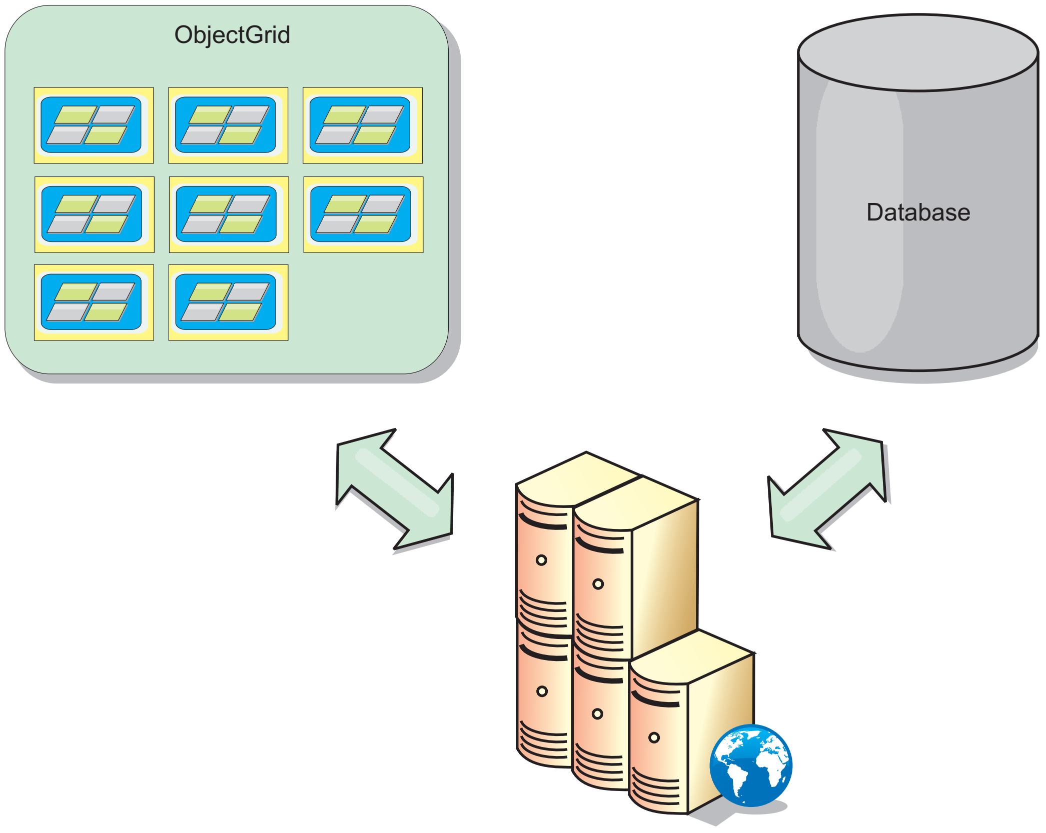


Figure 18. ObjectGrid as a side cache

Sparse and complete cache

WebSphere eXtreme Scale can be used as a sparse cache or a complete cache. A sparse cache only keeps a subset of the total data, while a complete cache keeps all of the data. and can be populated lazily, as the data is needed. Sparse caches are normally accessed using keys (instead of indexes or queries) because the data is only partially available.

Sparse cache

When a key is not present in a sparse cache, or the data is not available and a cache miss occurs, the next tier is invoked. The data is fetched, from a database for example, and is inserted into the data grid cache tier. If you are using a query or index, only the currently loaded values are accessed and the requests are not forwarded to the other tiers.

Complete cache

A complete cache contains all of the required data and can be accessed using non-key attributes with indexes or queries. A complete cache is preloaded with data from the database before the application tries to access the data. A complete cache can function as a database replacement after data is loaded. Because all of the data is available, queries and indexes can be used to find and aggregate data.

Side cache

When WebSphere eXtreme Scale is used as a side cache, the back end is used with the data grid.

Side cache

You can configure the product as a side cache for the data access layer of an application. In this scenario, WebSphere eXtreme Scale is used to temporarily store objects that would normally be retrieved from a back-end database. Applications check to see if the data grid contains the data. If the data is in the data grid, the data is returned to the caller. If the data does not exist, the data is retrieved from the back-end database. The data is then inserted into the data grid so that the next request can use the cached copy. The following diagram illustrates how WebSphere eXtreme Scale can be used as a side-cache with an arbitrary data access layer such as OpenJPA or Hibernate.

Side cache plug-ins for Hibernate and OpenJPA

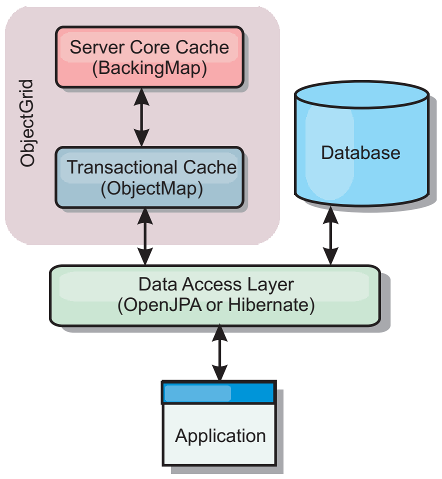

ObjectGrid
Server Core Cache (BackingMap)
Transactional Cache (ObjectMap)
Database
Data Access Layer (OpenJPA or Hibernate)
Application

Figure 19. Side cache

Cache plug-ins for both OpenJPA and Hibernate are included inWebSphere eXtreme Scale, so you can use the product as an automatic side-cache. Using WebSphere eXtreme Scale as a cache provider increases performance when reading and querying data and reduces load to the database. There are advantages thatWebSphere eXtreme Scale has over built-in cache implementations because the cache is automatically replicated between all processes. When one client caches a value, all other clients can use the cached value.

In-line cache

You can configure in-line caching for a database back end or as a side cache for a database. In-line caching uses eXtreme Scale as the primary means for interacting with the data. When eXtreme Scale is used as an in-line cache, the application interacts with the back end using a Loader plug-in.

In-line cache

When used as an in-line cache, WebSphere eXtreme Scale interacts with the back end using a Loader plug-in. This scenario can simplify data access because applications can access the eXtreme Scale APIs directly. Several different caching scenarios are supported in eXtreme Scale to make sure the data in the cache and the data in the back end are synchronized. The following diagram illustrates how an in-line cache interacts with the application and back end.

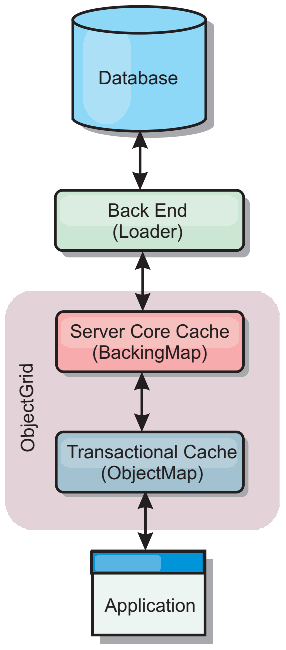


Figure 20. In-line cache

The in-line caching option simplifies data access because it allows applications to access the eXtreme Scale APIs directly. WebSphere eXtreme Scale supports several in-line caching scenarios, as follows.

- Read-through
- Write-through
- Write-behind

Read-through caching scenario

A read-through cache is a sparse cache that lazily loads data entries by key as they are requested. This is done without requiring the caller to know how the entries are populated. If the data cannot be found in the eXtreme Scale cache, eXtreme Scale will retrieve the missing data from the Loader plug-in, which loads the data from the back-end database and inserts the data into the cache. Subsequent requests for the same data key will be found in the cache until it is removed, invalidated or evicted.

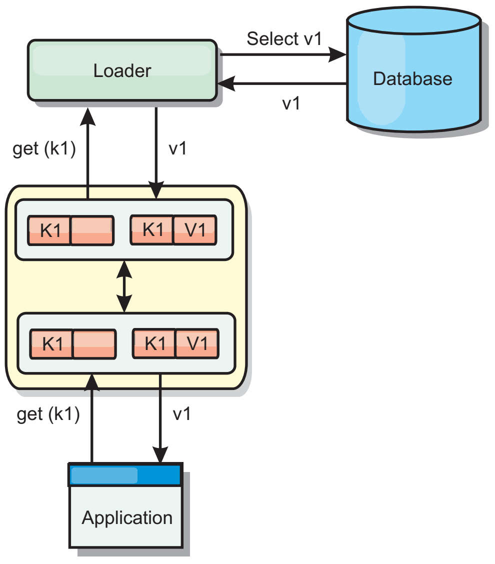


Figure 21. Read-through caching

Write-through caching scenario

In a write-through cache, every write to the cache synchronously writes to the database using the Loader. This method provides consistency with the back end, but decreases write performance since the database operation is synchronous. Since the cache and database are both updated, subsequent reads for the same data will be found in the cache, avoiding the database call. A write-through cache is often used in conjunction with a read-through cache.

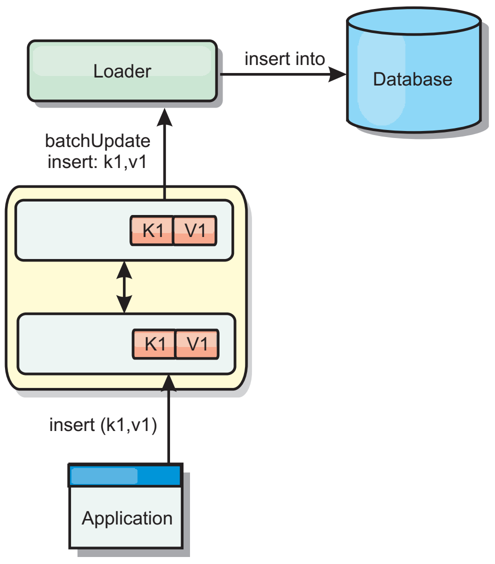


Figure 22. Write-through caching

Write-behind caching scenario

Database synchronization can be improved by writing changes asynchronously. This is known as a write-behind or write-back cache. Changes that would normally be written synchronously to the loader are instead buffered in eXtreme Scale and written to the database using a background thread. Write performance is significantly improved because the database operation is removed from the client transaction and the database writes can be compressed. See "Write-behind caching" for more information.

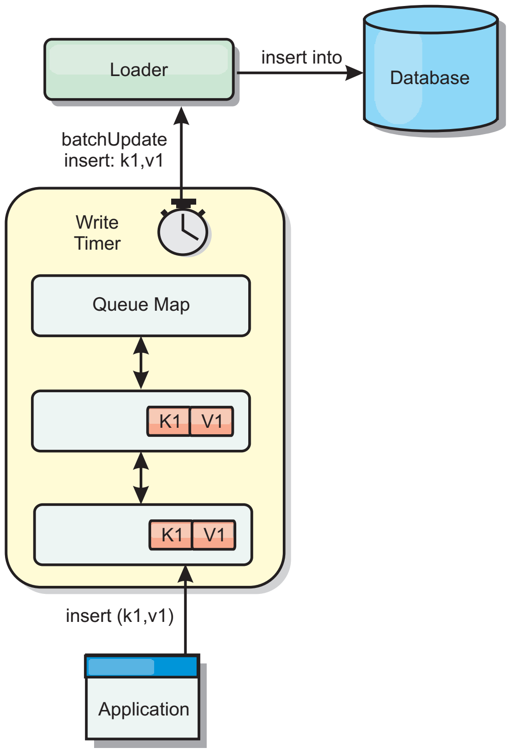


Figure 23. Write-behind caching

See "Write-behind caching" for further information.

Write-behind caching

You can use write-behind caching to reduce the overhead that occurs when updating a database you are using as a back end.

Write-behind caching overview

Write-behind caching asynchronously queues updates to the Loader plug-in. You can improve performance by disconnecting updates, inserts, and removes for a map, the overhead of updating the back-end database. The asynchronous update is performed after a time-based delay (for example, five minutes) or an entry-based delay (1000 entries).

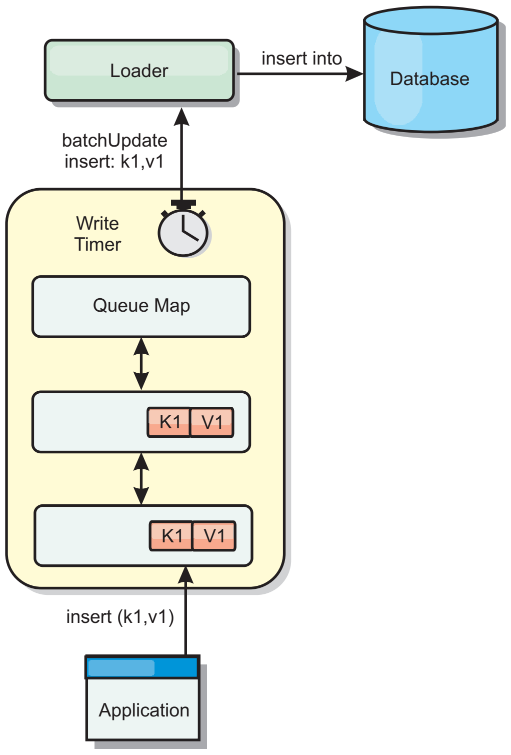


Figure 24. Write-behind caching

The write-behind configuration on a BackingMap creates a thread between the loader and the map. The loader then delegates data requests through the thread according to the configuration settings in the BackingMap.setWriteBehind method. When an eXtreme Scale transaction inserts, updates, or removes an entry from a map, a LogElement object is created for each of these records. These elements are sent to the write-behind loader and queued in a special ObjectMap called a queue map. Each backing map with the write-behind setting enabled has its own queue maps. A write-behind thread periodically removes the queued data from the queue maps and pushes them to the real back-end loader.

The write-behind loader only sends insert, update, and delete types of LogElement objects to the real loader. All other types of LogElement objects, for example, EVICT type, are ignored.

Benefits

Enabling write-behind support has the following benefits:

- **Back end failure isolation:** Write-behind caching provides an isolation layer from back end failures. When the back-end database fails, updates are queued in the queue map. The applications can continue driving transactions to eXtreme Scale. When the back end recovers, the data in the queue map is pushed to the back-end.
- **Reduced back end load:** The write-behind loader merges the updates on a key basis so only one merged update per key exists in the queue map. This merge decreases the number of updates to the back-end database.
- **Improved transaction performance:** Individual eXtreme Scale transaction times are reduced because the transaction does not need to wait for the data to be synchronized with the back-end.

Application design considerations

Enabling write-behind support is simple, but designing an application to work with write-behind support needs careful consideration. Without write-behind support, the ObjectGrid transaction encloses the back-end transaction. The ObjectGrid transaction starts before the back-end transaction starts, and it ends after the back-end transaction ends.

With write-behind support enabled, the ObjectGrid transaction finishes before the back-end transaction starts. The ObjectGrid transaction and back-end transaction are de-coupled.

Referential integrity constraints

Each backing map that is configured with write-behind support has its own write-behind thread to push the data to the back-end. Therefore, the data that updated to different maps in one ObjectGrid transaction are updated to the back-end in different back-end transactions. For example, transaction T1 updates key key1 in map Map1 and key key2 in map Map2. The key1 update to map Map1 is updated to the back-end in one back-end transaction, and the key2 updated to map Map2 is updated to the back-end in another back-end transaction by different write-behind threads. If data stored in Map1 and Map2 have relations, such as foreign key constraints in the back-end, the updates might fail.

When designing the referential integrity constraints in your back-end database, ensure that out-of-order updates are allowed.

Queue map locking behavior

Another major transaction behavior difference is the locking behavior. ObjectGrid supports three different locking strategies: PESSIMISTIC, OPTIMISITIC, and NONE. The write-behind queue maps uses pessimistic locking strategy no matter which lock strategy is configured for its backing map. Two different types of operations exist that acquire a lock on the queue map:

- When an ObjectGrid transaction commits, or a flush (map flush or session flush) happens, the transaction reads the key in the queue map and places an S lock on the key.
- When an ObjectGrid transaction commits, the transaction tries to upgrade the S lock to X lock on the key.

Because of this extra queue map behavior, you can see some locking behavior differences.

- If the user map is configured as PESSIMISTIC locking strategy, there isn't much locking behavior difference. Every time a flush or commit is called, an S lock is placed on the same key in the queue map. During the commit time, not only is an X lock acquired for key in the user map, it is also acquired for the key in the queue map.
- If the user map is configured as OPTIMISTIC or NONE locking strategy, the user transaction will follow the PESSIMISTIC locking strategy pattern. Every time a flush or commit is called, an S lock is acquired for the same key in the queue map. During the commit time, an X lock is acquired for the key in the queue map using the same transaction.

Loader transaction retries

ObjectGrid does not support 2-phase or XA transactions. The write-behind thread removes records from the queue map and updates the records to the back-end. If the server fails in the middle of the transaction, some back-end updates can be lost.

The write-behind loader will automatically retry to write failed transactions and will send an in-doubt LogSequence to the back-end to avoid data loss. This action requires the loader to be idempotent, which means when the Loader.batchUpdate(TxId, LogSequence) is called twice with the same value, it gives the same result as if it were applied one time. Loader implementations must implement the RetryableLoader interface to enable this feature. See the API documentation for more details.

Loader failures

The loader plug-in can fail when it is unable to communicate to the database back end. This can happen if the database server or the network connection is down. The write-behind loader will queue the updates and try to push the data changes to the loader periodically. The loader must notify the ObjectGrid run time that there is a database connectivity problem by throwing a LoaderNotAvailableException exception.

Therefore, the Loader implementation should be able to distinguish a data failure or a physical loader failure. Data failure should be thrown or re-thrown as a LoaderException or an OptimisticCollisionException, but a physical loader failure should be thrown or re-thrown as a LoaderNotAvailableException. ObjectGrid handles these two exceptions differently:

- If a LoaderException is caught by the write-behind loader, the write-behind loader will consider it fails due to some data failure, such as duplicate key failure. The write-behind loader will unbatch the update, and try the update one record at one time to isolate the data failure. If A {{LoaderException}}is caught again during the one record update, a failed update record is created and logged in the failed update map.
- If a LoaderNotAvailableException is caught by the write-behind loader, the write-behind loader will consider it fails because it cannot connect to the database end, for example, the database back-end is down, a database connection is not available, or the network is down. The write-behind loader will wait for 15 seconds and then re-try the batch update to the database.

The common mistake is to throw a LoaderException while a LoaderNotAvailableException should be thrown. All the records queued in the write-behind loader will become failed update records, which defeats the purpose of back-end failure isolation.

Performance considerations

Write-behind caching support increases response time by removing the loader update from the transaction. It also increases database throughput because database updates are combined. It is important to understand the overhead introduced by write-behind thread, which pulls the data out of the queue map and pushed to the loader.

The maximum update count or the maximum update time need to be adjusted based on the expected usage patterns and environment. If the value of the maximum update count or the maximum update time is too small, the overhead of the write-behind thread may exceed the benefits. Setting a large value for these two parameters could also increase the memory usage for queuing the data and increase the stale time of the database records.

For best performance, tune the write-behind parameters based on the following factors:

- Ratio of read and write transactions
- Same record update frequency
- Database update latency.

Loaders

With a Loader plug-in, a data grid map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or another 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 eXtreme Scale. A loader has the logic for reading and writing data to and from a persistent store.

Overview

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). The Loader is invoked when the cache is unable to satisfy a request for a key, providing read-through capability and lazy-population of the cache. A loader also allows updates to the database when cache values change. All changes within a transaction are grouped together to allow the number of database interactions to be minimized. A TransactionCallback plug-in is used in conjunction with the loader to trigger the demarcation of the backend transaction. Using this plug-in is important when multiple maps are included in a single transaction or when transaction data is flushed to the cache without committing.

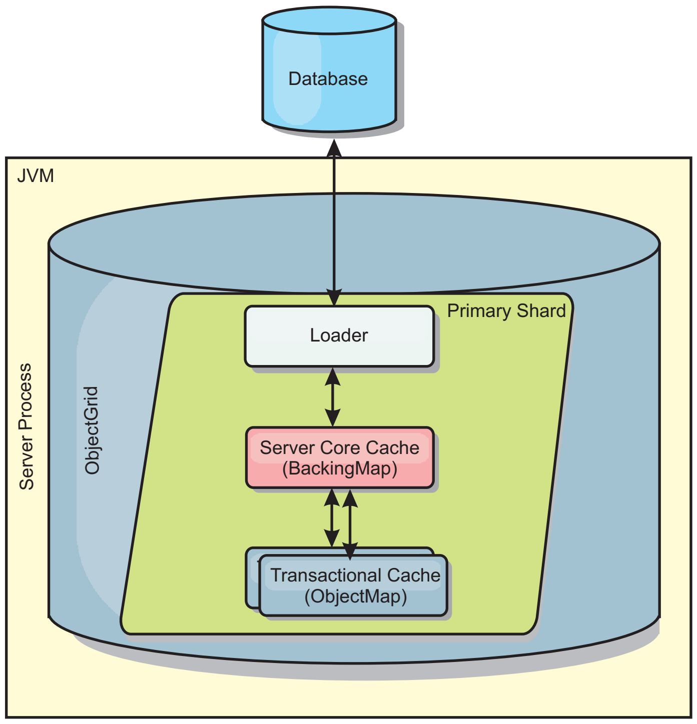


Figure 25. Loader

The loader can also use overqualified updates to avoid keeping database locks. By storing a version attribute in the cache value, the loader can see the before and after image of the value as it is updated in the cache. This value can then be used when updating the database or back end to verify that the data has not been updated. A Loader can also be configured to preload the data grid when it is started. When partitioned, a Loader instance is associated with each partition. If the "Company" Map has ten partitions, there are ten Loader instances, one per primary partition. When the primary shard for the Map is activated, the preloadMap method for the loader is invoked synchronously or asynchronously which allows loading the map partition with data from the back-end to occur automatically. When invoked synchronously, all client transactions are blocked, preventing inconsistent access to the data grid. Alternatively, a client preloader can be used to load the entire data grid.

Two built-in loaders can greatly simplify integration with relational database back ends. The JPA loaders utilize the Object-Relational Mapping (ORM) capabilities of both the OpenJPA and Hibernate implementations of the Java Persistence API (JPA) specification. See the information about JPA loaders in the *Product Overview* for more information.

Loader configuration

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.

For more information about writing and configuring loaders, see the *Programming Guide*.

Data pre-loading and warm-up

In many scenarios that incorporate the use of a loader, you can prepare your data grid by pre-loading it with data.

When used as a complete cache, the data grid must hold all of the data and must be loaded before any clients can connect to it. When you are using a sparse cache, you can warm up the cache with data so that clients can have immediate access to data when they connect.

Two approaches exist for pre-loading data into the data grid: Using a Loader plug-in or using a client loader, as described in the following sections.

Loader plug-in

The loader plug-in is associated with each map and is responsible for synchronizing a single primary partition shard with the database. The preloadMap method of the loader plug-in is invoked automatically when a shard is activated. For example, if you have 100 partitions, 100 loader instances exist, each loading the data for its partition. When run synchronously, all clients are blocked until the preload has completed.

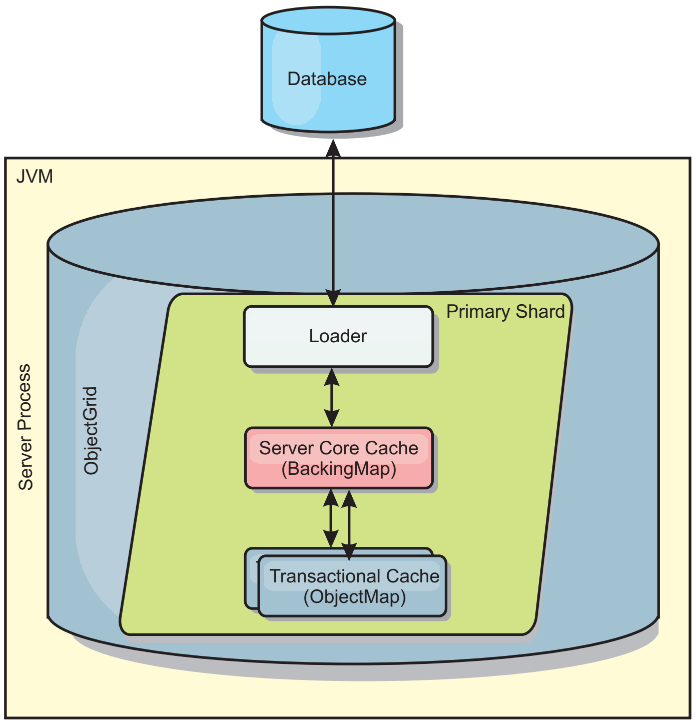

Database
JVM
Server Process
ObjectGrid
Primary Shard
Loader
Server Core Cache (BackingMap)
Transactional Cache (ObjectMap)

Figure 26. Loader plug-in

See the details on using a loader in the *Programming Guide* for more information.

Client loader

A client loader is a pattern for using one or more clients to load the grid with data. Using multiple clients to load grid data can be effective when the partition scheme is not stored in the database. You can invoke client loaders manually or automatically when the data grid starts. Client loaders can optionally use the StateManager to set the state of the data grid to pre-load mode, so that clients are not able to access the grid while it is pre-loading the data. WebSphere eXtreme Scale includes a Java Persistence API (JPA)-based loader that you can use to automatically load the data grid with either the OpenJPA or Hibernate JPA providers. For more information about cache providers, see “JPA cache plug-in” on page 61.

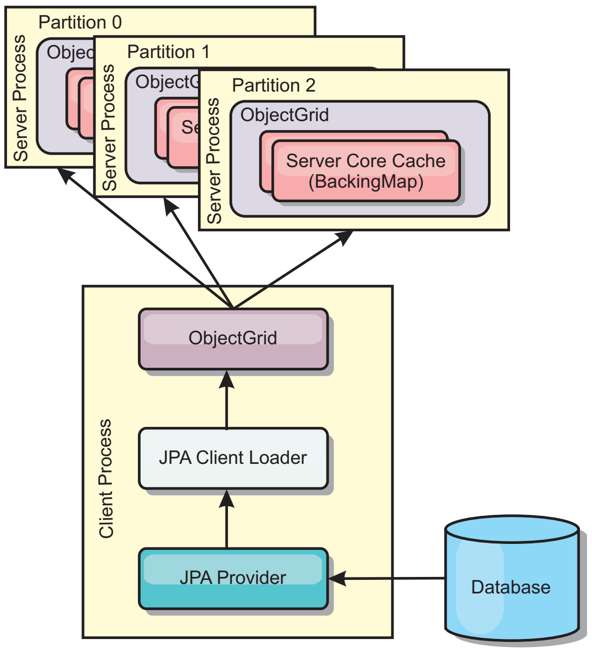


Figure 27. Client loader

Map pre-loading

Maps can be associated with Loaders. A loader is used to fetch objects when they cannot be found in the map (a cache miss) and also to write changes to a back-end when a transaction commits. Loaders can also be used for preloading data into a map. The preloadMap method of the Loader interface is called on each map when its corresponding partition in the MapSet becomes a primary. The preloadMap method is not called on replicas. It attempts to load all the intended referenced data from the back-end into the map using the provided session. The relevant map is identified by the BackingMap argument that is passed to the preloadMap method.

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

Preloading in partitioned MapSet

Maps can be partitioned into N partitions. Maps can therefore be striped across multiple servers, with each entry identified by a key that is stored only on one of those servers. Very large maps can be held in an eXtreme Scale because the application is no longer limited by the heap size of a single JVM to hold all the entries of a Map. Applications that want to preload with the preloadMap method of the Loader interface must identify the subset of the data that it preloads. A fixed number of partitions always exists. You can determine this number by using the following code example:

```
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();
int myPartition = backingMap.getPartitionId();
```

This code example shows how an application can identify the subset of the data to preload from the database. Applications must always use these methods even when the map is not initially partitioned. These methods allow flexibility: If the map is later partitioned by the administrators, then the loader continues to work correctly.

The application must issue queries to retrieve the *myPartition* subset from the backend. If a database is used, then it might be easier to have a column with the partition identifier for a given record unless there is some natural query that allows the data in the table to partition easily.

See details on writing a loader with a replica preload controller in the *Programming Guide* for an example on how to implement a Loader for a replicated eXtreme Scale.

Performance

The preload implementation copies data from the back-end into the map by storing multiple objects in the map in a single transaction. The optimal number of records to store per transaction depends on several factors, including complexity and size. For example, after the transaction includes blocks of more than 100 entries, the performance benefit decreases as you increase the number of entries. To determine the optimal number, begin with 100 entries and then increase the number until the performance benefit decreases to none. Larger transactions result in better replication performance. Remember, only the primary runs the preload code. The preloaded data is replicated from the primary to any replicas that are online.

Preloading MapSets

If the application uses a MapSet with multiple maps then each map has its own loader. Each loader has a preload method. Each map is loaded serially by the eXtreme Scale. It might be more efficient to preload all the maps by designating a single map as the preloading map. This process is an application convention. For example, two maps, department and employee, might use the department Loader to preload both the department and the employee maps. This procedure ensures that, transactionally, if an application wants a department then the employees for that department are in the cache. When the department Loader preloads a department from the back-end, it also fetches the employees for that department. The department object and its associated employee objects are then added to the map using a single transaction.

Recoverable preloading

Some customers have very large data sets that need caching. Preloading this data can be very time consuming. Sometimes, the preloading must complete before the application can go online. You can benefit from making preloading recoverable. Suppose there are a million records to preload. The primary is preloading them and fails at the 800,000th record. Normally, the replica chosen to be the new primary clears any replicated state and starts from the beginning. eXtreme Scale can use a ReplicaPreloadController interface. The loader for the application would also need to implement the ReplicaPreloadController interface. This example adds a single method to the Loader: `Status checkPreloadStatus(Session session, BackingMap bmap);`. This method is called by the eXtreme Scale run time before the preload method of the Loader interface is normally called. The eXtreme Scale tests the result of this method (Status) to determine its behavior whenever a replica is promoted to a primary.

Table 3. Status value and response

Returned status value	**eXtreme Scale response**
Status.PRELOADED_ALREADY	eXtreme Scale does not call the preload method at all because this status value indicates that the map is fully preloaded.
Status.FULL_PRELOAD_NEEDED	eXtreme Scale clears the map and calls the preload method normally.
Status.PARTIAL_PRELOAD_NEEDED	eXtreme Scale leaves the map as-is and calls preload. This strategy allows the application loader to continue preloading from that point onwards.

Clearly, while a primary is preloading the map, it must leave some state in a map in the MapSet that is being replicated so that the replica determines what status to return. You can use an extra map named, for example, RecoveryMap. This RecoveryMap must be part of the same MapSet that is being preloaded to ensure that the map is replicated consistently with the data being preloaded. A suggested implementation follows.

As the preload commits each block of records, the process also updates a counter or value in the RecoveryMap as part of that transaction. The preloaded data and the RecoveryMap data are replicated atomically to the replicas. When the replica is promoted to primary, it can now check the RecoveryMap to see what has happened.

The RecoveryMap can hold a single entry with the state key. If no object exists for this key then you need a full preload (checkPreloadStatus returns FULL_PRELOAD_NEEDED). If an object exists for this state key and the value is COMPLETE, the preload completes, and the checkPreloadStatus method returns PRELOADED_ALREADY. Otherwise, the value object indicates where the preload restarts and the checkPreloadStatus method returns PARTIAL_PRELOAD_NEEDED. The loader can store the recovery point in an instance variable for the loader so that when preload is called, the loader knows the starting point. The RecoveryMap can also hold an entry per map if each map is preloaded independently.

Handling recovery in synchronous replication mode with a Loader

The eXtreme Scale run time is designed not to lose committed data when the primary fails. The following section shows the algorithms used. These algorithms apply only when a replication group uses synchronous replication. A loader is optional.

The eXtreme Scale run time can be configured to replicate all changes from a primary to the replicas synchronously. When a synchronous replica is placed, it receives a copy of the existing data on the primary shard. During this time, the primary continues to receives transactions and copies them to the replica asynchronously. The replica is not considered to be online at this time.

After the replica catches up the primary, the replica enters peer mode and synchronous replication begins. Every transaction committed on the primary is sent to the synchronous replicas and the primary waits for a response from each replica. A synchronous commit sequence with a Loader on the primary looks like the following set of steps:

Table 4. Commit sequence on the primary

Step with loader	**Step without loader**
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same
Send changes to replicas and wait for acknowledgement	same
Commit to the loader through the TransactionCallback plug-in	plug-in commit called, but does nothing
Release locks for entries	same

Notice that the changes are sent to the replica before they are committed to the loader. To determine when the changes are committed on the replica, revise this sequence: At initialize time, initialize the tx lists on the primary as below.

```
CommitedTx = {}, RolledBackTx = {}
```

During synchronous commit processing, use the following sequence:

Table 5. Synchronous commit processing

Step with loader	**Step without loader**
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same
Send changes with a committed transaction, roll back transaction to replica, and wait for acknowledgement	same
Clear list of committed transactions and rolled back transactions	same
Commit the loader through the TransactionCallBack plug-in	TransactionCallBack plug-in commit is still called, but typically does not do anything
If commit succeeds, add the transaction to the committed transactions, otherwise add to the rolled back transactions	no-op
Release locks for entries	same

For replica processing, use the following sequence:

1. Receive changes
2. Commit all received transactions in the committed transaction list
3. Roll back all received transactions in the rolled back transaction list
4. Start a transaction or session
5. Apply changes to the transaction or session
6. Save the transaction or session to the pending list
7. Send back reply

Notice that on the replica, no loader interactions occur while the replica is in replica mode. The primary must push all changes through the Loader. The replica does not make any changes. A side effect of this algorithm is that the replica always has the transactions, but they are not committed until the next primary transaction sends the commit status of those transactions. The transactions are then committed or rolled back on the replica. Until then, the transactions are not committed. You can add a timer on the primary that sends the transaction outcome after a small period of time (a few seconds). This timer limits, but does not eliminate, any staleness to that time window. This staleness is only a problem when using replica read mode. Otherwise, the staleness does not have an impact on the application.

When the primary fails, it is likely that a few transactions were committed or rolled back on the primary, but the message never made it to the replica with these outcomes. When a replica is promoted to the new primary, one of the first actions is to handle this condition. Each pending transaction is reprocessed against the new primary's set of maps. If there is a Loader, then each transaction is given to the Loader. These transactions are applied in strict first in first out (FIFO) order. If a transactions fails, it is ignored. If three transactions are pending, A, B, and C, then A might commit, B might rollback and C might also commit. No one transaction has any impact on the others. Assume that they are independent.

A loader might want to use slightly different logic when it is in failover recovery mode versus normal mode. The loader can easily know when it is in failover recovery mode by implementing the ReplicaPreloadController interface. The checkPreloadStatus method is only called when failover recovery completes. Therefore, if the apply method of the Loader interface is called before the checkPreloadStatus method, then it is a recovery transaction. After the checkPreloadStatus method is called, the failover recovery is complete.

Database synchronization techniques

When WebSphere eXtreme Scale is used as a cache, applications must be written to tolerate stale data if the database can be updated independently from an eXtreme Scale transaction. To serve as a synchronized in-memory database processing space, eXtreme Scale provides several ways of keeping the cache updated.

Database synchronization techniques

Periodic refresh

The cache can be automatically invalidated or updated periodically using the Java Persistence API (JPA) time-based database updater.The updater periodically queries the database using a JPA provider for any updates or inserts that have occurred since the previous update. Any changes identified are automatically invalidated or updated when used with a sparse cache. If used with a complete cache, the entries can be discovered and inserted into the cache. Entries are never removed from the cache.

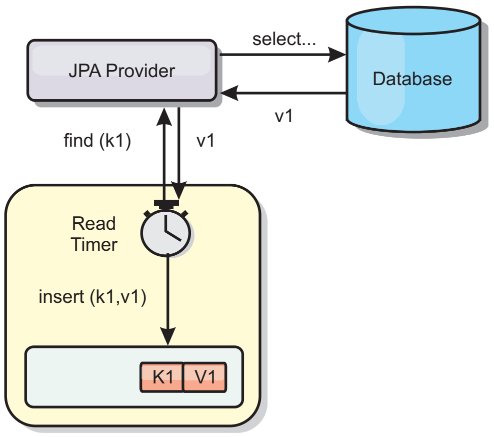


Figure 28. Periodic refresh

Eviction

Sparse caches can utilize eviction policies to automatically remove data from the cache without affecting the database. There are three built-in policies included in eXtreme Scale: time-to-live, least-recently-used, and least-frequently-used. All three policies can optionally evict data more aggressively as memory becomes constrained by enabling the memory-based eviction option.

Event-based invalidation

Sparse and complete caches can be invalidated or updated using an event generator such as Java Message Service (JMS). Invalidation using JMS can be manually tied to any process that updates the back-end using a database trigger. A JMS ObjectGridEventListener plug-in is provided in eXtreme Scale that can notify clients when the server cache has any changes. This can decrease the amount of time the client can see stale data.

Programmatic invalidation

The eXtreme Scale APIs allow manual interaction of the near and server cache using the Session.beginNoWriteThrough(), ObjectMap.invalidate() and EntityManager.invalidate() API methods. If a client or server process no longer needs a portion of the data, the invalidate methods can be used to remove data from the near or server cache. The beginNoWriteThrough method applies any ObjectMap or EntityManager operation to the local cache without calling the loader. If invoked from a client, the operation applies only to the near cache (the remote loader is not invoked). If invoked on the server, the operation applies only to the server core cache without invoking the loader.

Invalidating stale cache data

To reduce the window of time when clients may see stale data, you can use an event-based or programmatic invalidation mechanism.

Event-based invalidation

Sparse and complete caches can be invalidated or updated using an event generator such as Java Message Service (JMS). Invalidation using JMS can be manually tied to any process that updates the back-end using a database trigger. A JMS ObjectGridEventListener plug-in is provided in eXtreme Scale that can notify clients when the server cache changes. This type of notification decreases the amount of time the client can see stale data.

Event-based invalidation normally consists of the following three components.

- **Event queue:** An event queue stores the data change events. It could be a JMS queue, a database, an in-memory FIFO queue, or any kind of manifest as long as it can manage the data change events.
- **Event publisher:** An event publisher publishes the data change events to the event queue. An event publisher is usually an application you create or an eXtreme Scale plug-in implementation. The event publisher knows when the data is changed or it changes the data itself. When a transaction commits, events are generated for the changed data and the event publisher publishes these events to the event queue.
- **Event consumer:** An event consumer consumes data change events. The event consumer is usually an application to ensure the target grid data is updated with the latest change from other grids. This event consumer interacts with the event queue to get the latest data change and applies the data changes in the target grid. The event consumers can use eXtreme Scale APIs to invalidate stale data or update the grid with the latest data.

For example, JMSObjectGridEventListener has an option for a client-server model, in which the event queue is a designated JMS destination. All server processes are event publishers. When a transaction commits, the server gets the data changes and publishes them to the designated JMS destination. All the client processes are event consumers. They receive the data changes from the designated JMS destination and apply the changes to the client's near cache.

See the topic on enabling the client invalidation mechanism in the *Administration Guide* for more information.

Programmatic invalidation

The WebSphere eXtreme Scale APIs allow manual interaction of the near and server cache using the Session.beginNoWriteThrough(), ObjectMap.invalidate() and EntityManager.invalidate() API methods. If a client or server process no longer needs a portion of the data, the invalidate methods can be used to remove data from the near or server cache. The beginNoWriteThrough method applies any ObjectMap or EntityManager operation to the local cache without calling the loader. If invoked from a client, the operation applies only to the near cache (the remote loader is not invoked). If invoked on the server, the operation applies only to the server core cache without invoking the loader.

You can use programmatic invalidation with other techniques to determine when to invalidate the data. For example, this invalidation method uses event-based invalidation mechanisms to receive the data change events, and then uses APIs to invalidate the stale data.

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.

The HashIndex class is the built-in index plug-in implementation that can support both of the built-in application index interfaces: MapIndex and MapRangeIndex. You also can create your own indexes. You can add HashIndex as either a static or dynamic index into the BackingMap, obtain either MapIndex or MapRangeIndex index proxy object, and use the index proxy object to find cached objects.

For information about configuring the HashIndex, refer to Configuring the HashIndex.

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.

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.

Java object caching concepts

WebSphere eXtreme Scale is primarily used as a data grid and cache for Java objects. You can use several APIs to interact with the eXtreme Scale grid to access and store these objects.

This topic describes some of the common APIs and some of the concepts that you must be aware of when choosing an API and deployment topology. See the “Caching architecture: Maps, containers, clients, and catalogs” on page 11 topic for a description of the various services and topologies that eXtreme Scale provides.

WebSphere eXtreme Scale's central component is the ObjectGrid. The ObjectGrid is the namespace that stores related data, and contains sets of hash maps, each holding key-value pairs. These maps can be grouped together and partitioned and made highly available and scalable.

Because the grid holds Java objects by nature, there are some important considerations when designing an application so that the grid can store and access data efficiently. Factors that can affect scalability, performance and memory utilization include the following.

Class loader and classpath considerations

Since eXtreme Scale stores Java objects in the cache by default, you must define classes on the classpath wherever the data is accessed.

Specifically, eXtreme Scale client and container processes must include the classes or JARs in the classpath when starting the process. When designing an application for use with eXtreme Scale, separate out any business logic from the persistent data objects.

See Class loading in the WebSphere Application Server Network Deployment information center for more information.

For considerations within a Spring Framework setting, see the packaging section under the topic on integrating with Spring framework in the *Programming Guide*.

For settings related to using the WebSphere eXtreme Scale instrumentation agent, see the instrumentation agent topic in the *Programming Guide*.

Relationship management

Object-oriented languages such as Java, and relational databases support relationships or associations. Relationships decrease the amount of storage through the use of object references or foreign keys.

When you are using relationships in a data grid, the data must be organized in a constrained tree. One root type must exist in the tree and all children must be associated to only one root. For example: Department can have many Employees and an Employee can have many Projects. But a Project cannot have many Employees that belong to different departments. Once a root is defined, all access to that root object and its descendants are managed through the root. WebSphere eXtreme Scale uses the hash code of the root object's key to choose a partition. For example:

```
partition = (hashCode MOD numPartitions).
```

When all of the data for a relationship is tied to a single object instance, the entire tree can be collocated in a single partition and can be accessed very efficiently using one transaction. If the data spans multiple relationships, then multiple partitions must be involved which involves additional remote calls, which can lead to performance bottlenecks.

Reference data

Some relationships include look-up or reference data such as: `CountryName`. With look-up or reference data, the data should exist in every partition. The data can be accessed by any root key and the same result is returned. Reference data such as this should only be used in cases where the data is fairly static. Updating this data can be expensive because the data needs to be updated in every partition. The DataGrid API is a common technique to keeping reference data up-to-date.

Costs and benefits of normalization

Normalizing the data using relationships can help reduce the amount of memory used by the data grid since duplication of data is decreased. However, in general, the more relational data that is added, the less it will scale out. When data is grouped together, it becomes more expensive to maintain the relationships and to keep the sizes manageable. Since the grid partitions data based on the key of the root of the tree, the size of the tree isn't taken into account. Therefore, if you have a lot of relationships for one tree instance, the data grid may become unbalanced, causing one partition to hold more data than the others.

When the data is denormalized or flattened, the data that would normally be shared between two objects is instead duplicated and each table can be partitioned independently, providing a much more balanced data grid. Although this increases the amount of memory used, it allows the application to scale since a single row of data can be accessed that has all of the necessary data. This is ideal for read-mostly grids since maintaining the data becomes more expensive.

For more information, see Classifying XTP systems and scaling.

Managing relationships using the data access APIs

The ObjectMap API is the fastest, most flexible and granular of the data access APIs, providing a transactional, session-based approach at accessing data in the grid of maps. The ObjectMap API allows clients to use common CRUD (create, read, update and delete) operations to manage key-value pairs of objects in the distributed data grid.

When using the ObjectMap API, object relationships must be expressed by embedding the foreign key for all relationships in the parent object.

An example follows.

```
public class Department {
 Collection<String> employeeIds;
}
```

The EntityManager API simplifies relationship management by extracting the persistent data from the objects including the foreign keys. When the object is later retrieved from the data grid, the relationship graph is rebuilt, as in the following example.

```
@Entity
public class Department {
 Collection<String> employees;
}
```

The EntityManager API is very similar to other Java object persistence technologies such as JPA and Hibernate in that it synchronizes a graph of managed Java object instances with the persistent store. In this case, the persistent store is an eXtreme Scale data grid, where each entity is represented as a map and the map contains the entity data rather than the object instances.

Cache key considerations

WebSphere eXtreme Scale uses hash maps to store data in the grid, where a Java object is used for the key.

Guidelines

When choosing a key, consider the following requirements:

- Keys can never change. If a portion of the key needs to change, then the cache entry should be removed and reinserted.
- Keys should be small. Since keys are used in every data access operation, it's a good idea to keep the key small so that it can be serialized efficiently and use less memory.
- Implement a good hash and equals algorithm. The hashCode and equals(Object o) methods must always be overridden for each key object.
- Cache the key's hashCode. If possible, cache the hash code in the key object instance to speed up hashCode() calculations. Since the key is immutable, the hashCode should be cacheable.
- Avoid duplicating the key in the value. When using the ObjectMap API, it is convenient to store the key inside the value object. When this is done, the key data is duplicated in memory.

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 Plug-ins for serializing and copying cached objects and ObjectTransformer interface best practices.

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.

See the CopyMode method best practices in the *Programming Guide* for more information.

Inserting data for different time zones

When inserting data with calendar, java.util.Date, and timestamp attributes into an ObjectGrid, you must ensure these date time attributes are created based on same time zone, especially when deployed into multiple servers in various time zones. Using the same time zone based date time objects can ensure the application is time-zone safe and data can be queried by calendar, java.util.Date and timestamp predicates.

Without explicitly specifying a time zone when creating date time objects, Java will use the local time zone and may cause inconsistent date time values in clients and servers.

Consider an example in a distributed deployment in which client1 is in time zone [GMT-0] and client2 is in [GMT-6] and both want to create a java.util.Date object with value '1999-12-31 06:00:00'. Then client1 will create java.util.Date object with value '1999-12-31 06:00:00 [GMT-0]' and client2 will create java.util.Date object with value '1999-12-31 06:00:00 [GMT-6]'. Both java.util.Date objects are not equal because the time zone is different. A similar problem occurs when preloading data into partitions residing in servers in different time zones if local time zone is used to create date time objects.

To avoid the described problem, the application can choose a time zone such as [GMT-0] as the base time zone for creating calendar, java.util.Date, and timestamp objects.

For more information, see the topic on querying data in multiple time zones in the *Programming Guide.*

Chapter 3. Cache integration overview

The crucial element that gives WebSphere eXtreme Scale the capability to perform with such versatility and reliability is its application of caching concepts to optimize the persistence and recollection of data in virtually any deployment environment.

JPA Loaders

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.

You can use a Java Persistence API (JPA) loader plug-in implementation with eXtreme Scale to interact with any database supported by your chosen loader. 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.

The JPALoader com.ibm.websphere.objectgrid.jpa.JPALoader and the JPAEntityLoader com.ibm.websphere.objectgrid.jpa.JPAEntityLoader plug-ins are two built-in JPA loader plug-ins that are used to synchronize the ObjectGrid maps with a database. You must have a JPA implementation, such as Hibernate or OpenJPA, to use this feature. The database can be any back end that is supported by the chosen JPA provider.

You can 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.

JPA loader architecture

The JPA Loader is used for eXtreme Scale maps that store plain old Java objects (POJO).

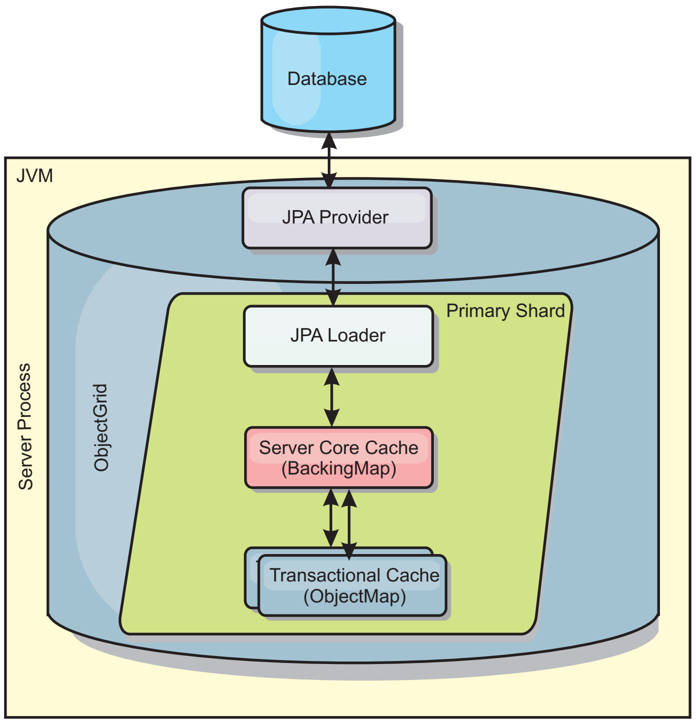


Figure 29. JPA Loader architecture

When an ObjectMap.get(Object key) method is called, the eXtreme Scale run time first checks whether the entry is contained in the ObjectMap layer. If not, the run time delegates the request to the JPA Loader. Upon request of loading the key, the JPALoader calls the JPA EntityManager.find(Object key) method to find the data from the JPA layer. If the data is contained in the JPA entity manager, it is returned; otherwise, the JPA provider interacts with the database to get the value.

When an update to ObjectMap occurs, for example, using the ObjectMap.update(Object key, Object value) method, the eXtreme Scale run time creates a LogElement for this update and sends it to the JPALoader. The JPALoader calls the JPA EntityManager.merge(Object value) method to update the value to the database.

For the JPAEntityLoader, the same four layers are involved. However, because the JPAEntityLoader plug-in is used for maps that store eXtreme Scale entities, relations among entities could complicate the usage scenario. An eXtreme Scale entity is distinguished from JPA entity. For more information, see the information about the JPAEntityLoader plug-in in the *Programming Guide*.

Methods

Loaders provide three main methods:

1. get: Returns a list of values that correspond to the list of keys that are passed in by retrieving the data using JPA. The method uses JPA to find the entities in the database. For the JPALoader plug-in, the returned list contains a list of JPA entities directly from the find operation. For the JPAEntityLoader plug-in, the returned list contains eXtreme Scale entity value tuples that are converted from the JPA entities.
2. batchUpdate: Writes the data from ObjectGrid maps to the database. Depending on different operation types (insert, update, or delete), the loader

uses the JPA persist, merge, and remove operations to update the data to the database. For the JPALoader, the objects in the map are directly used as JPA entities. For the JPAEntityLoader, the entity tuples in the map are converted into objects which are used as JPA entities.

3. preloadMap: Preloads the map using the ClientLoader.load client loader method. For partitioned maps, the preloadMap method is only called in one partition. The partition is specified the preloadPartition property of the JPALoader or JPAEntityLoader class. If the preloadPartition value is set to less than zero, or greater than (*total_number_of_partitions* - 1), preload is disabled.

Both JPALoader and JPAEntityLoader plug-ins work with the JPATxCallback class to coordinate the eXtreme Scale transactions and JPA transactions. JPATxCallback must be configured in the ObjectGrid instance to use these two loaders.

Configuration and programming

For more information about configuring JPA loaders, see the information about JPA loaders in the *Administration Guide*. For more information about programming JPA loaders, see the*Programming Guide*.

JPA cache plug-in

WebSphere eXtreme Scale includes level 2 (L2) cache plug-ins for both OpenJPA and Hibernate Java Persistence API (JPA) providers.

Using eXtreme Scale as an L2 cache provider increases performance when you are reading and querying data and reduces load to the database. WebSphere eXtreme Scale has advantages over built-in cache implementations because the cache is automatically replicated between all processes. When one client caches a value, all other clients are able to use the cached value that is locally in-memory.

With the OpenJPA and Hibernate ObjectGrid cache plug-ins, you can create three topology types: embedded, embedded-partitioned, and remote.

Embedded topology

An embedded topology creates an eXtreme Scale server within the process space of each application. OpenJPA and Hibernate read the in-memory copy of the cache directly and write to all of the other copies. You can improve the write performance by using asynchronous replication. This default topology performs best when the amount of cached data is small enough to fit in a single process.

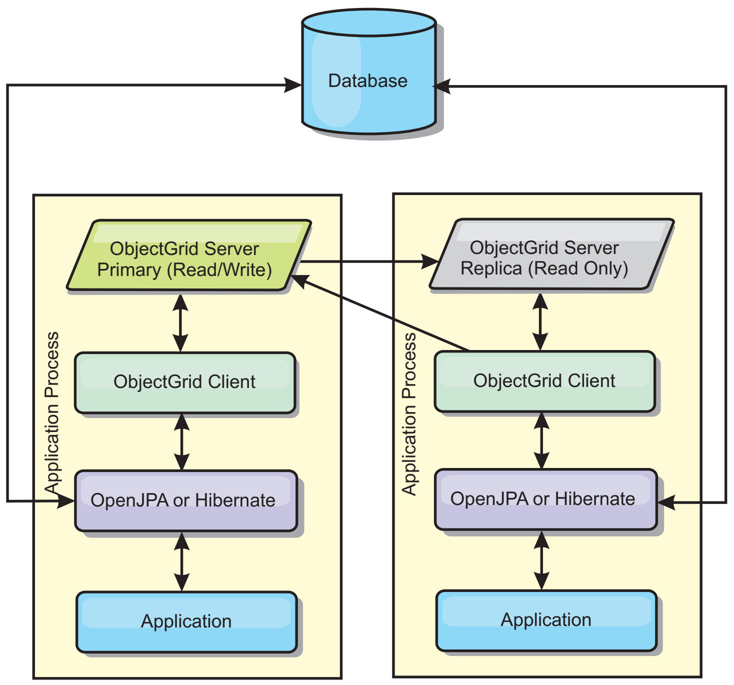


Figure 30. JPA embedded topology

Advantages:
- All cache reads are very fast, local accesses.
- Simple to configure.

Limitations:
- Amount of data is limited to the size of the process.
- All cache updates are sent to one process.

Embedded, partitioned topology

When the cached data is too large to fit in a single process, the embedded, partitioned topology uses ObjectGrid partitions to divide the data over multiple processes. Performance is not as high as the embedded topology because most cache reads are remote. However, you can still use this option when database latency is high.

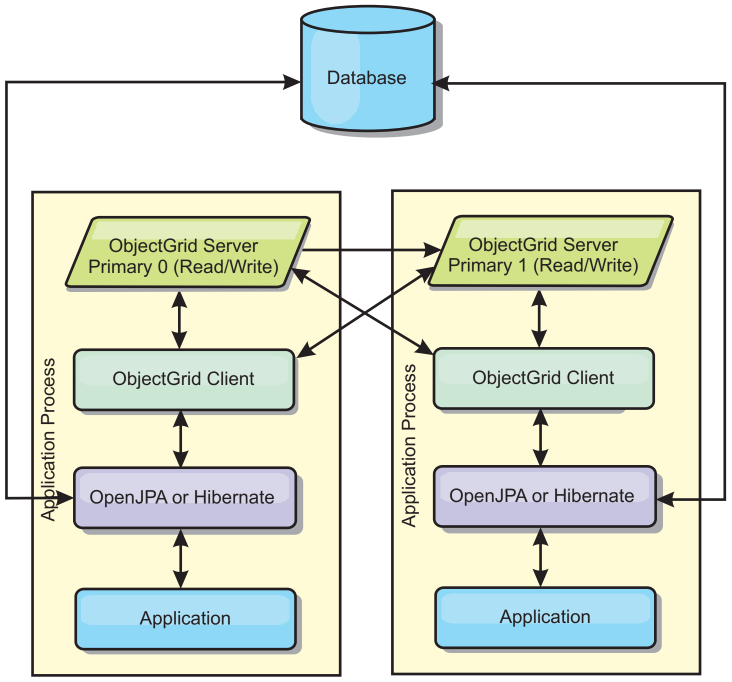


Figure 31. JPA embedded, partitioned topology

Advantages:
- Stores large amounts of data.
- Simple to configure
- Cache updates are spread over multiple processes.

Limitation:
- Most cache reads and updates are remote.

For example, to cache 10 GB of data with a maximum of 1 GB per JVM, ten Java virtual machines are required. The number of partitions must therefore be set to 10 or more. Ideally, the number of partitions should be set to a prime number where each shard stores a reasonable amount of memory. Usually, the numberOfPartitions setting is equal to the number of Java virtual machines. With this setting, each JVM stores one partition. If you enable replication, you must increase the number of Java virtual machines in the system. Otherwise, each JVM also stores one replica partition, which consumes as much memory as a primary partition.

Read about sizing memory and partition count calculation in the *Administration Guide* to maximize the performance of your chosen configuration.

For example, in a system with 4 Java virtual machines, and the numberOfPartitions setting value of 4, each JVM hosts a primary partition. A read operation has a 25 percent chance of fetching data from a locally available partition, which is much faster compared to getting data from a remote JVM. If a read operation, such as running a query, needs to fetch a collection of data that involves 4 partitions evenly, 75 percent of the calls are remote and 25 percent of the calls are local. If the ReplicaMode setting is set to either `SYNC` or `ASYNC` and the ReplicaReadEnabled setting is set to `true`, then four replica partitions are created and spread across four Java virtual machines. Each JVM hosts one primary partition and one replica partition. The chance that the read operation runs locally increases to 50 percent. The read operation that fetches a collection of data that involves four partitions evenly has 50 percent remote calls and 50 percent local calls. Local calls are much faster than remote calls. Whenever remote calls occur, the performance drops.

Remote topology

A remote topology stores all of the cached data in one or more separate processes, reducing memory use of the application processes. You can take advantage of distributing your data over separate processes by deploying a partitioned, replicated eXtreme Scale data grid. As opposed to the embedded and embedded partitioned configurations described in the previous sections, if you want to manage the remote data grid, you must do so independent of the application and JPA provider. Read about monitoring your deployment environment for more information on managing an eXtreme Scale data grid deployment.

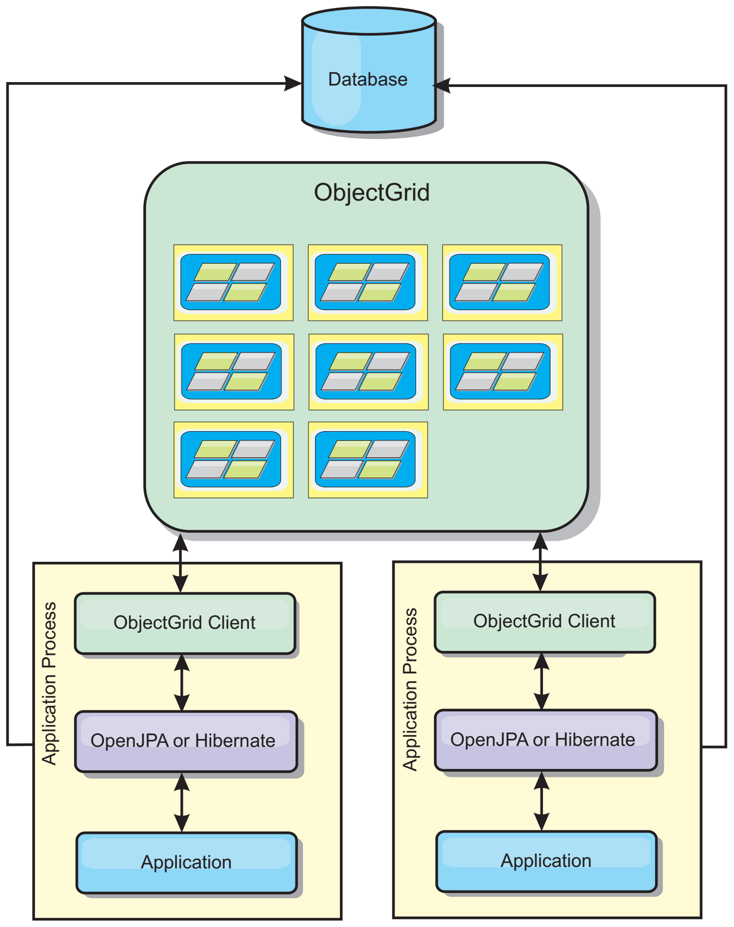


Figure 32. JPA remote topology

Advantages:
- Stores large amounts of data.
- Application process is free of cached data.
- Cache updates are spread over multiple processes.
- Very flexible configuration options.

Limitation:
- All cache reads and updates are remote.

Configuration

For more information about configuring the JPA cache plug-ins, see the plug-ins section in the *Programming Guide*.

HTTP session management

The session replication manager that is shipped with WebSphere eXtreme Scale can work with the default session manager in the application server to replicate session data from one process to another process to support user session data high availability.

Features

The session manager has been designed so that it can run in any Java Platform, Enterprise Edition Version 1.4 container. Because the session manager does not have any dependencies on WebSphere APIs, it can support various versions of WebSphere Application Server, as well as vendor application server environments.

The HTTP session manager provides session replication capabilities for an associated application. The session replication manager works with the web container's session manager to create HTTP sessions and manage the life cycles of HTTP sessions that are associated with the application. These life cycle management activities include: the invalidation of sessions based on a timeout or an explicit servlet or JavaServer Pages (JSP) call and the invocation of session listeners that are associated with the session or the web application. The session manager persists its sessions in an ObjectGrid instance. This instance can be a local, in-memory instance or a fully replicated, clustered and partitioned instance. The use of the latter topology enables the session manager to provide HTTP session failover support when application servers are shut down or end unexpectedly. The session manager can also work in environments that do not support affinity, when affinity is not enforced by a load balancer tier that sprays requests to the application server tier.

Usage scenarios

The session manager can be used in the following scenarios:
- In environments that use application servers at different versions of WebSphere Application Server, such as in a classic migration scenario.
- In deployments that use application servers from different vendors. For example, an application that is being developed on open source application servers and that is hosted on WebSphere Application Server. Another example is an application that is being promoted from staging to production. Seamless migration of these application server versions is possible while all HTTP sessions are live and being serviced.
- In environments that require the user to persist sessions with higher quality of service (QoS) levels and better guarantees of session availability during server failover than default WebSphere Application Server QoS levels.
- In an environment where session affinity cannot be guaranteed, or environments in which affinity is maintained by a vendor load balancer and the affinity mechanism must be customized to that load balancer.
- In an environment to offload the overhead of session management and storage to an external Java process.
- In multiple cells to enable session failover between cells.

- In multiple data centers or multiple zones.

How the session manager works

The session replication manager uses standard session listener to listen on the changes of session data, and persists the session data into an ObjectGrid instance either locally or remotely. The session data is reloaded in the request path through the standard servlet from the ObjectGrid instance either locally or remotely. You can add the session listener and servlet filter to every web module in your application with tooling that ships with WebSphere eXtreme Scale. You can also manually add these listeners and filters to the web deployment descriptor of your application.

This session replication manager works with each vendor's web container session manager to replicate session data across Java virtual machines. When the original server dies, users can retrieve session data from other servers.

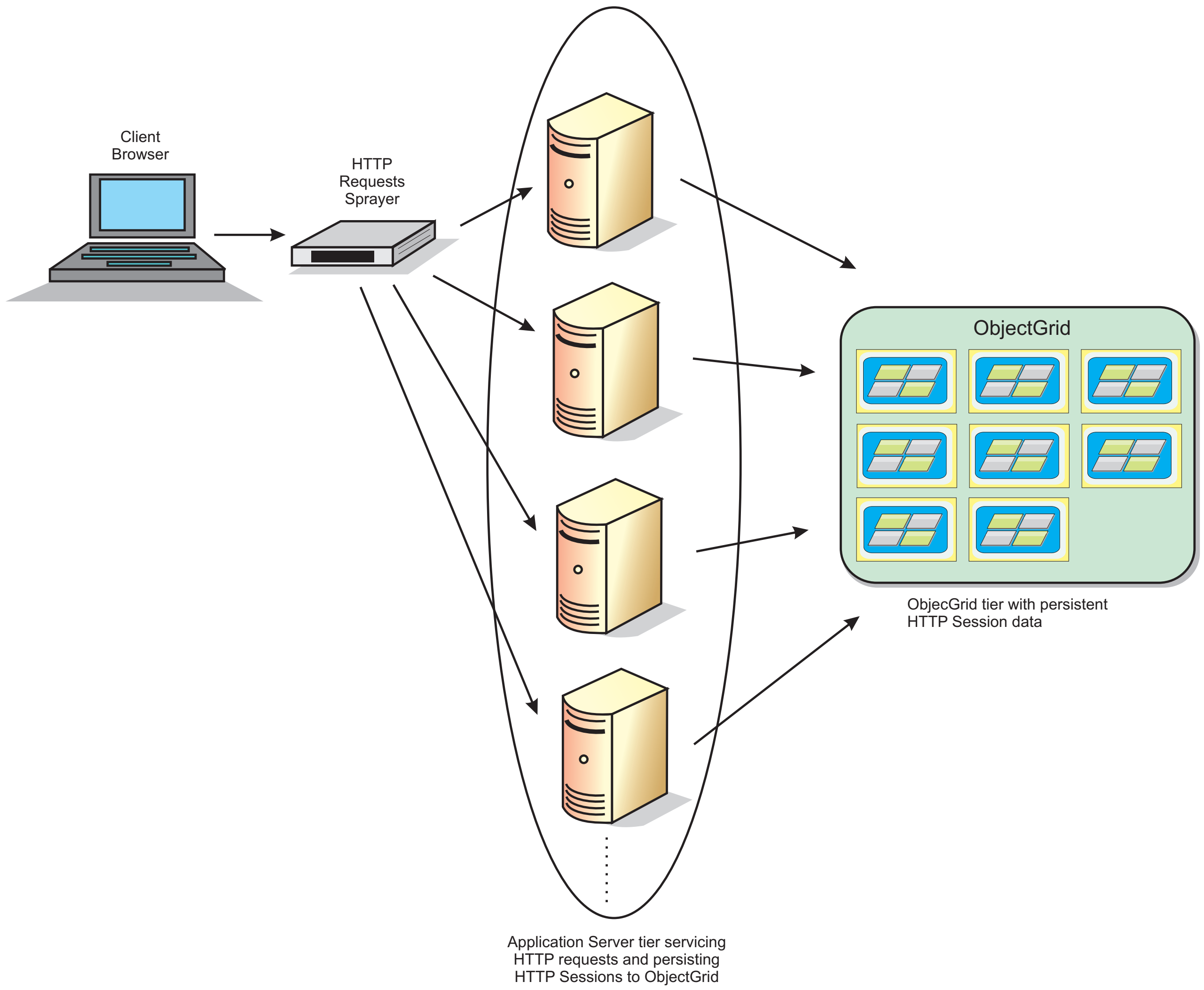

Client Browser
HTTP Requests Sprayer
ObjectGrid
ObjecGrid tier with persistent HTTP Session data
Application Server tier servicing HTTP requests and persisting HTTP Sessions to ObjectGrid

Figure 33. HTTP session management topology with a remote container configuration

Deployment topologies

The session manager can be configured using two different dynamic deployment scenarios:

- **Embedded, network attached eXtreme Scale containers**

  In this scenario, the eXtreme Scale servers are collocated in the same processes as the servlets. The session manager can communicate directly to the local ObjectGrid instance, avoiding costly network delays. This scenario is preferable when running with affinity and performance is critical
- **Remote, network attached eXtreme Scale containers**

  In this scenario, the eXtreme Scale servers run in external processes the process in which the servlets run. The session manager communicates with a remote eXtreme Scale server grid. This scenario is preferable when the web container tier does not have the memory to store the session data. The session data will be offloaded to a separate tier, which will result in lower memory usage on the web container tier, but higher latency due to the remote location of the data.

Generic embedded container startup

eXtreme Scale automatically starts an embedded ObjectGrid container inside any application-server process when the web container initializes the session listener or servlet filter, if the objectGridType property is set to EMBEDDED. See Servlet context initialization parameters for details.

You are not required to package an ObjectGrid.xml file and objectGridDeployment.xml file into your web application WAR or EAR file with eXtreme Scale Version 7.1. The default ObjectGrid.xml and objectGridDeployment.xml files are packaged in the product JAR. Dynamic maps are created for various web application contexts by default. Static eXtreme Scale maps continue to be supported.

This approach for starting embedded ObjectGrid containers applies to any type of application server. The approaches involving aWebSphere Application Server component or WebSphere Application Server Community Edition GBean are deprecated.

Listener-based session replication manager

The eXtreme Scale session replication manager that is shipped with WebSphere eXtreme Scale can work with the default session manager in the application server to replicate session data from one process to another process to support user session data high availability.

The session manager has been designed so that it can run in any Java™ Platform, Enterprise Edition Version 1.4 container. The session manager does not have any dependencies on WebSphere APIs, so it is capable of supporting various versions of WebSphere Application Server as well as vendor application server environments.

The HTTP session manager provides session replication capabilities for an associated application. The session replication manager works with web container's session manager to create HTTP sessions and manage the life cycles of HTTP sessions that are associated with the application. These life cycle management activities include: the invalidation of sessions based on a timeout or an explicit servlet or JavaServer Pages (JSP) call and the invocation of session listeners that are associated with the session or the web application. The session manager persists its sessions in an ObjectGrid instance. This instance can be a local, in-memory instance or a fully replicated, clustered and partitioned instance. The use of the latter topology allows the session manager to provide HTTP session failover support when application servers are shut down or end unexpectedly. The session manager can also work in environments that do not support affinity, when affinity is not enforced by a load balancer tier that sprays requests to the application server tier.

Usage scenarios

The session manager can be used in the following scenarios:

- In environments that use application servers at different versions of WebSphere Application Server, such as in a classic migration scenario.
- In deployments that use application servers from different vendors. For example, an application that is being developed on open source application servers and that are hosted on WebSphere Application Server. Another example is an application that is being promoted from staging to production. Seamless migration of these application server versions is possible while all HTTP sessions are live and being serviced.
- In environments that require the user to persist sessions with higher quality of service (QoS) levels and better guarantees of session availability during server failover than default WebSphere Application Server QoS levels.
- In an environment where session affinity cannot be guaranteed, or environments in which affinity is maintained by a vendor load balancer and the affinity mechanism needs to be customized to that load balancer.
- In an environment to offload the overhead of session management and storage to an external Java process.
- In multiple cells to enable session failover between cells.
- In multiple data centers or multiple zones.

Session manager details

The session replication manager uses standard session listener to listen on the changes of session data, and persists the session data into an ObjectGrid instance either locally or remotely. The session data is reloaded in the request path through the standard servlet from the ObjectGrid instance either locally or remotely. You can add the session listener and servlet filter to every Web module in your application with tooling that ships with WebSphere eXtreme Scale. You can also manually add these listeners and filters to the Web deployment descriptor of your application.

The session replication manager works with each vendor's base session manager to replicate application session data. Note the following considerations.

- Choose embedded ObjectGrid containers or remote ObjectGrid containers, depending on performance requirements and your data sizes. The embedded scenario gives the easiest configuration and better performance.
- Choose the number of remote ObjectGrid containers per web container according to user data sizes.
- Choose to store the whole session data together or store each attribute separately, depending on the number and size of attributes user data and change frequencies.
- Choose the replication interval. Less aggressive replication interval gives better performance.

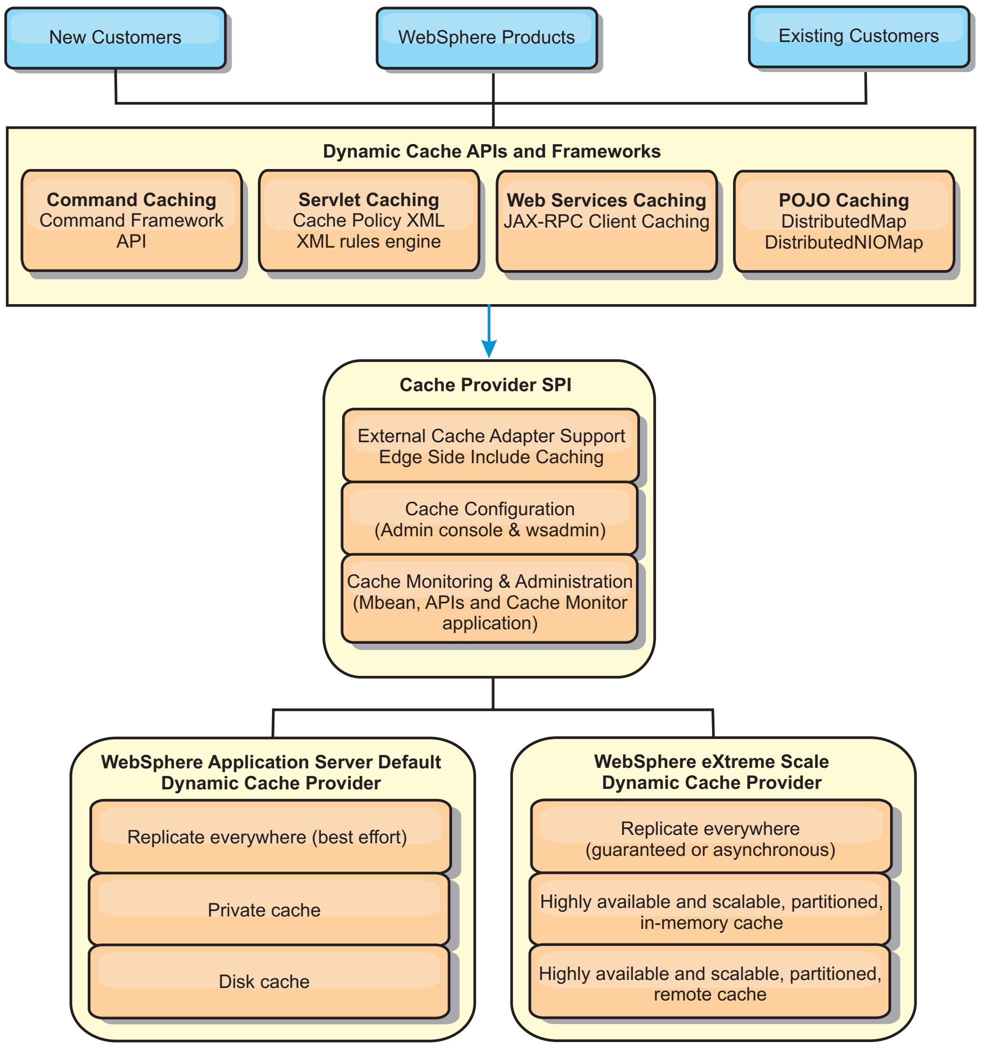
- Choose the session table size to balance local memory size and performance. The product offloads session user data when local session cache maximum size is reached.
- The product supports HTTP sessions within the application context, according to the Servlet specification, to avoid security and use attribute naming conflict issues. You can share sessions across application context by their singleton class.
- The session replication manager replicates session data for high availability, by listening to the session data changes and reloading the stored session data on demand. This is accomplished by reusing the web container base session manager of each vendor. The session is linked together through various native session IDs. Session data is failed over from one web container into another web container by reloading session data from ObjectGrid instance. Session ID and creation time could differ before and after failover.

Dynamic cache provider

The Dynamic Cache API is available to Java EE applications that are deployed in WebSphere Application Server. The dynamic cache provider can be leveraged to cache business data, generated HTML, or to synchronize the cached data in the cell by using the data replication service (DRS).

Overview

Previously, the only service provider for the Dynamic Cache API was the default dynamic cache engine built into WebSphere Application Server. Customers can use the dynamic cache service provider interface in WebSphere Application Server to plug eXtreme Scale into dynamic cache. By setting up this capability, you can enable applications written with the Dynamic Cache API or applications using container-level caching (such as servlets) to leverage the features and performance capabilities of WebSphere eXtreme Scale.


New Customers
WebSphere Products
Existing Customers
Dynamic Cache APIs and Frameworks
Command Caching
Command Framework API
Servlet Caching
Cache Policy XML
XML rules engine
Web Services Caching
JAX-RPC Client Caching
POJO Caching
DistributedMap
DistributedNIOMap
Cache Provider SPI
External Cache Adapter Support
Edge Side Include Caching
Cache Configuration
(Admin console & wsadmin)
Cache Monitoring & Administration
(Mbean, APIs and Cache Monitor application)
WebSphere Application Server Default Dynamic Cache Provider
Replicate everywhere (best effort)
Private cache
Disk cache
WebSphere eXtreme Scale Dynamic Cache Provider
Replicate everywhere (guaranteed or asynchronous)
Highly available and scalable, partitioned, in-memory cache
Highly available and scalable, partitioned, remote cache

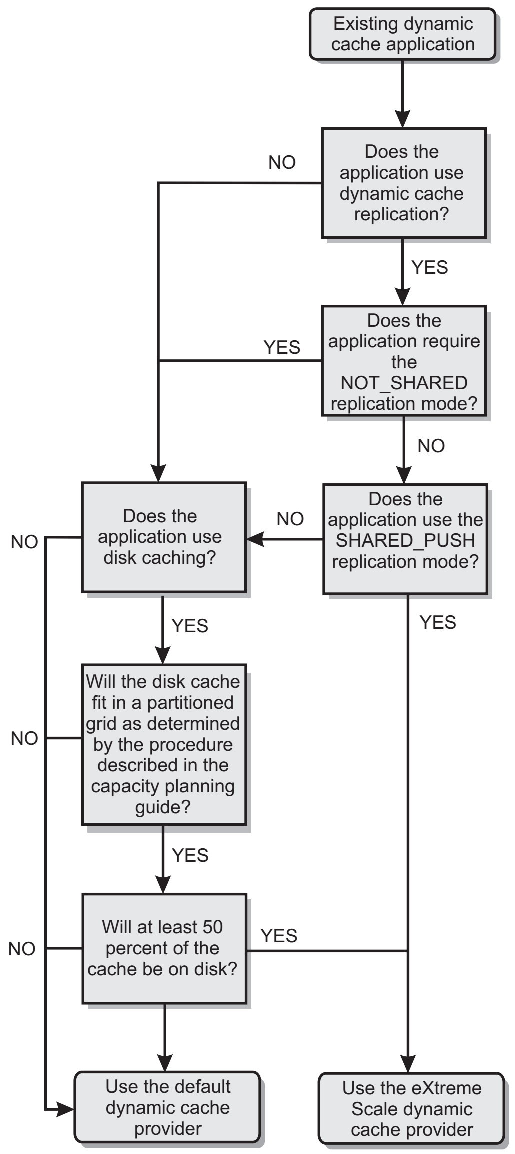
You can install and configure the dynamic cache provider as described in Configuring the dynamic cache provider for WebSphere eXtreme Scale.

Deciding how to leverage WebSphere eXtreme Scale

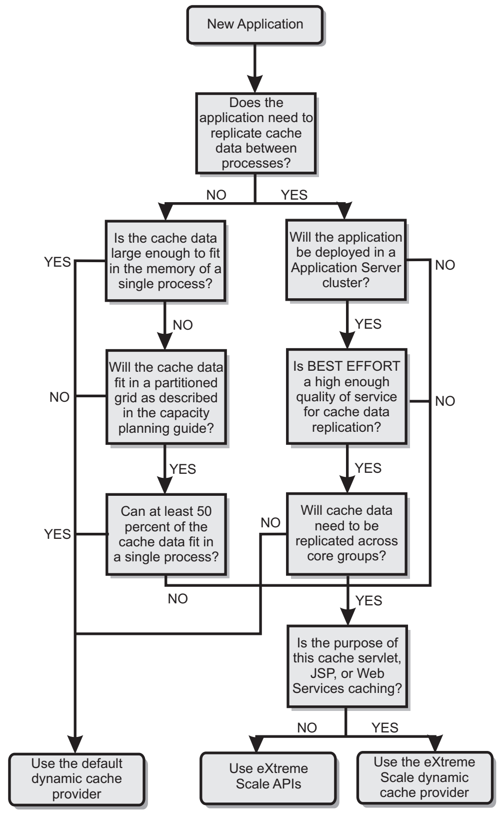
The available features in WebSphere eXtreme Scale significantly increase the distributed capabilities of the Dynamic Cache API beyond what is offered by the default dynamic cache engine and data replication service. With eXtreme Scale, you can create caches that are truly distributed between multiple servers, rather than just replicated and synchronized between the servers. Also, eXtreme Scale caches are transactional and highly available, ensuring that each server sees the same contents for the dynamic cache service. WebSphere eXtreme Scale offers a higher quality of service for cache replication than DRS.

However, these advantages do not mean that the eXtreme Scale dynamic cache provider is the right choice for every application. Use the decision trees and feature comparison matrix below to determine what technology fits your application best.

Decision tree for migrating existing dynamic cache applications


Existing dynamic cache application
Does the application use dynamic cache replication?
NO
YES
Does the application require the NOT_SHARED replication mode?
YES
NO
Does the application use the SHARED_PUSH replication mode?
NO
YES
Does the application use disk caching?
NO
YES
Will the disk cache fit in a partitioned grid as determined by the procedure described in the capacity planning guide?
NO
YES
Will at least 50 percent of the cache be on disk?
NO
YES
Use the default dynamic cache provider
Use the eXtreme Scale dynamic cache provider

Decision tree for choosing a cache provider for new applications


New Application
Does the application need to replicate cache data between processes?
NO
YES
Is the cache data large enough to fit in the memory of a single process?
YES
NO
Will the cache data fit in a partitioned grid as described in the capacity planning guide?
NO
YES
Can at least 50 percent of the cache data fit in a single process?
YES
NO
Will the application be deployed in a Application Server cluster?
NO
YES
Is BEST EFFORT a high enough quality of service for cache data replication?
NO
YES
Will cache data need to be replicated across core groups?
NO
YES
Is the purpose of this cache servlet, JSP, or Web Services caching?
NO
YES
Use the default dynamic cache provider
Use eXtreme Scale APIs
Use the eXtreme Scale dynamic cache provider

Feature comparison

Table 6. Feature comparison

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
Local, in-memory caching	x	x	x
Distributed caching	Embedded	Embedded, embedded-partitioned and remote-partitioned	Multiple
Linearly scalable		x	x
Reliable replication (synchronous)		ORB	ORB
Disk overflow	x		
Eviction	LRU/TTL/heap-based	LRU/TTL (per partition)	Multiple
Invalidation	x	x	x
Relationships	Dependency IDs, templates	Dependency IDs, templates	x
Non-key lookups			Query and index
Back-end integration			Loaders
Transactional		Implicit	x

Table 6. Feature comparison (continued)

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
Key-based storage	x	x	x
Events and listeners	x	x	x
WebSphere Application Server integration	Single cell only	Multiple cell	Cell independent
Java Standard Edition support		x	x
Monitoring and statistics	x	x	x
Security	x	x	x

Table 7. Seamless technology integration

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
WebSphere Application Server servlet/JSP results caching	V5.1+	V6.1.0.25+	
WebSphere Application Server Web Services (JAX-RPC) result caching	V5.1+	V6.1.0.25+	
HTTP session caching			x
Cache provider for OpenJPA and Hibernate			x
Database synchronization using OpenJPA and Hibernate			x

Table 8. Programming interfaces

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
Command-based API	Command framework API	Command framework API	DataGrid API
Map-based API	DistributedMap API	DistributedMap API	ObjectMap API
EntityManager API			x

For a more detailed description on how eXtreme Scale distributed caches work, see the deployment configuration information in the *Administration Guide*.

Note: An eXtreme Scale distributed cache can only store entries where the key and the value both implement the java.io.Serializable interface.

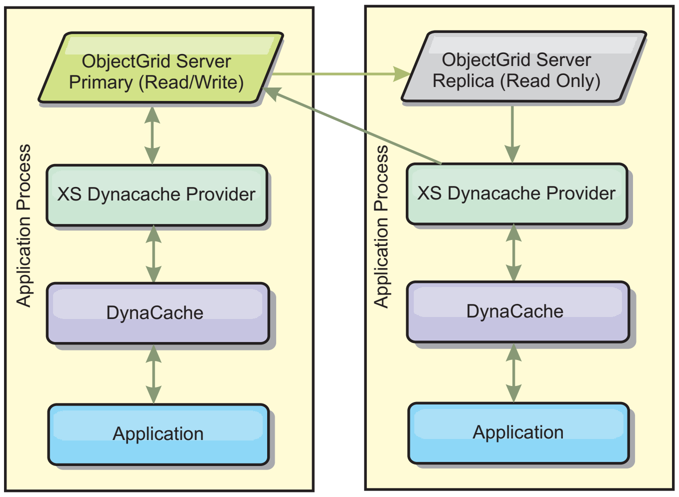
Topology types

A dynamic cache service created with the eXtreme Scale provider can be deployed in any of three available topologies, allowing you to tailor the cache specifically to performance, resource, and administrative needs. These topologies are embedded, embedded partitioned, and remote.

Embedded topology

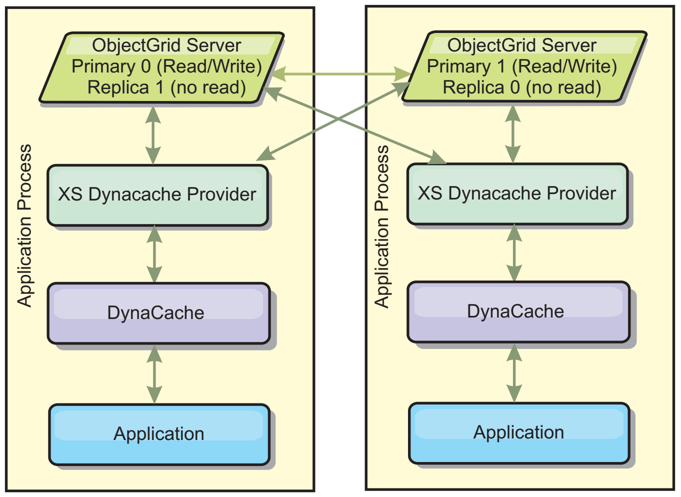
The embedded topology is similar to the default dynamic cache and DRS provider. Distributed cache instances created with the embedded topology keep a full copy of the cache within each eXtreme Scale process that accesses the dynamic cache service, allowing all read operations to occur locally. All write operations go through a single-server process, in which the transactional locks are managed, before being replicated to the rest of the servers. Consequently, this topology is better for workloads where cache-read operations greatly outnumber cache-write operations.

With the embedded topology, new or updated cache entries are not immediately visible on every single server process. A cache entry will not be visible, even to the server that generated it, until it propagates through the asynchronous replication services of WebSphere eXtreme Scale. These services operate as fast as the hardware will allow, but there is still a small delay. The embedded topology is shown in the following image:



Embedded partitioned topology

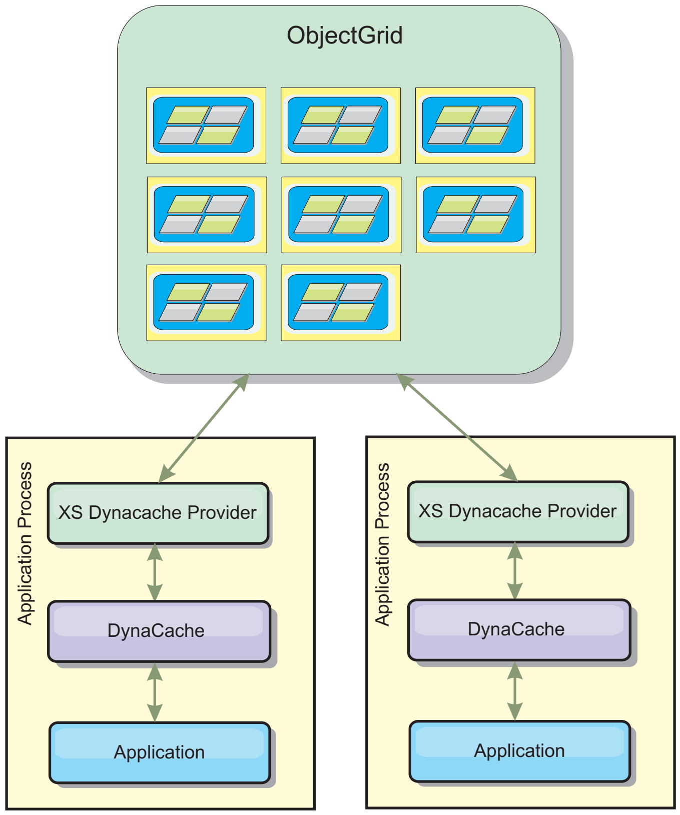
For workloads where cache-writes occur as often as or more frequently than reads, the embedded partitioned or remote topologies are recommended. The embedded partitioned topology keeps all of the cache data within the WebSphere Application Server processes that access the cache. However, each process only stores a portion of the cache data. All reads and writes for the data located on this “partition” go through the process, meaning that most requests to the cache will be fulfilled with a remote procedure call. This results in a higher latency for read operations than the embedded topology, but the capacity of the distributed cache to handle read and write operations will scale linearly with the number of WebSphere Application Server processes accessing the cache. Also, with this topology, the maximum size of the cache is not bound by the size of a single WebSphere process. Because each process only holds a portion of the cache, the maximum cache size becomes the aggregate size of all the processes, minus the overhead of the process. The embedded partitioned topology is shown in the following image:



For example, assume you have a grid of server processes with 256 megabytes of free heap each to host a dynamic cache service. The default dynamic cache provider and the eXtreme Scale provider using the embedded topology would both be limited to an in-memory cache size of 256 megabytes minus overhead. See the Capacity Planning and High Availability section later in this document. The eXtreme Scale provider using the embedded partitioned topology would be limited to a cache size of one gigabyte minus overhead. In this manner, the WebSphere eXtreme Scale provider makes it possible to have an in-memory dynamic cache services that are larger than the size of a single server process. The default dynamic cache provider relies on the use of a disk cache to allow cache instances to grow beyond the size of a single process. In many situations, the WebSphere eXtreme Scale provider can eliminate the need for a disk cache and the expensive disk storage systems needed to make them perform.

Remote topology

The remote topology can also be used to eliminate the need for a disk cache. The only difference between the remote and embedded partitioned topologies is that all of the cache data is stored outside of WebSphere Application Server processes when you are using the remote topology. WebSphere eXtreme Scale supports standalone container processes for cache data. These container processes have a lower overhead than a WebSphere Application Server process and are also not limited to using a particular Java Virtual Machine (JVM). For example, the data for a dynamic cache service being accessed by a 32-bit WebSphere Application Server process could be located in an eXtreme Scale container process running on a 64-bit JVM. This allows users to leverage the increased memory capacity of 64-bit processes for caching, without incurring the additional overhead of 64-bit for application server processes. The remote topology is shown in the following image:



Data compression

Another performance feature offered by the WebSphere eXtreme Scale dynamic cache provider that can help users manage cache overhead is compression. The default dynamic cache provider does not allow for compression of cached data in memory. With the eXtreme Scale provider, this becomes possible. Cache compression using the deflate algorithm can be enabled on any of the three distributed topologies. Enabling compression will increase the overhead for read and write operations, but will drastically increase cache density for applications like servlet and JSP caching.

Local in-memory cache

The WebSphere eXtreme Scale dynamic cache provider can also be used to back dynamic cache instances that have **replication disabled**. Like the default dynamic cache provider, these caches can store non-serializable data. They can also offer better performance than the default dynamic cache provider on large multi-processor enterprise servers because the eXtreme Scale code path is designed to maximize in-memory cache concurrency.

Dynamic cache engine and eXtreme Scale functional differences

In the case of local in-memory caches where replication is disabled, there should be no appreciable functional difference between caches backed by the default dynamic cache provider and WebSphere eXtreme Scale. Users should not notice a functional difference between the two caches except that the WebSphere eXtreme Scale backed caches do not support disk offload or statistics and operations related to the size of the cache in memory.

In the case of caches where replication is enabled there will be no appreciable difference in the results returned by most Dynamic Cache API calls, regardless of whether the customer is using the default dynamic cache provider or the eXtreme Scale dynamic cache provider. For some operations you cannot emulate the behavior of the dynamic cache engine using eXtreme Scale.

Dynamic cache statistics

Dynamic cache statistics are reported via the CacheMonitor application or the dynamic cache MBean. When using the eXtreme Scale dynamic cache provider, statistics will still be reported through these interfaces, but the context of the statistical values will be different.

If a dynamic cache instance is shared between three servers named A, B, and C, then the dynamic cache statistics object only returns statistics for the copy of the cache on the server where the call was made. If the statistics are retrieved on server A, they only reflect the activity on server A.

With eXtreme Scale, there is only a single distributed cache shared among all the servers, so it is not possible to track most statistics on a server-by-server basis like the default dynamic cache provider does. A list of the statistics reported by the Cache Statistics API and what they represent when you are using the WebSphere eXtreme Scale dynamic cache provider follows. Like the default provider, these statistics are not synchronized and therefore can vary up to 10% for concurrent workloads.

- **Cache Hits** : Cache hits are tracked per server. If traffic on Server A generates 10 cache hits and traffic on Server B generates 20 cache hits, the cache statistics will report 10 cache hits on Server A and 20 cache hits on Server B.
- **Cache Misses**: Cache misses are tracked per server just like cache hits.
- **Memory Cache Entries**: This statistic reports the number of cache entries in the distributed cache. Every server that accesses the cache will report the same value for this statistic, and that value will be the total number of cache entries in memory over all the servers.
- **Memory Cache Size in MB**: This metric is supported only for caches using the remote, embedded, or embedded_partitioned topologies. It reports the number of megabytes of Java heap space consumed by the cache, across the entire grid. This statistic reports heap usage only for the primary partitions; you must take replicas into account. Because the default setting for the remote and embedded_partitioned topologies is one asynchronous replica, double this number to get the true memory consumption of the cache.
- **Cache Removes**: This statistic reports the total number of entries removed from the cache by any method, and is an aggregate value for the whole distributed cache. If traffic on Server A generates 10 invalidations and traffic on Server B generates 20 invalidations, then the value on both servers will be 30.
- **Cache Least Recently Used (LRU) Removes**: This statistic is aggregate, like cache removes. It tracks the number of entries that were removed to keep the cache under its maximum size.
- **Timeout Invalidations**: This is also an aggregate statistic, and it tracks the number of entries that were removed because they timed out.
- **Explicit Invalidations** : Also an aggregate statistic, this tracks the number of entries that were removed with direct invalidation by key, dependency ID or template.
- **Extended Stats** : The eXtreme Scale dynamic cache provider exports the following extended stat key strings.

- **com.ibm.websphere.xs.dynacache.remote_hits**: The total number of cache hits tracked at the eXtreme Scale container. This is an aggregate statistic, and its value in the extended stats map is a `long`.
- **com.ibm.websphere.xs.dynacache.remote_misses**: The total number of cache misses tracked at the eXtreme Scale container. An aggregate statistic, its value in the extended stats map is a `long`.

Reporting reset statistics

The dynamic cache provider allows you to reset cache statistics. With the default provider the reset operation only clears the statistics on the affected server. The eXtreme Scale dynamic cache provider tracks most of its statistical data on the remote cache containers. This data is not cleared or changed when the statistics are reset. Instead the default dynamic cache behavior is simulated on the client by reporting the difference between the current value of a given statistic and the value of that statistic the last time reset was called on that server.

For example, if traffic on Server A generates 10 cache removes, the statistics on Server A and on Server B will report 10 removes. Now, if the statistics on Server B are reset and traffic on Server A generates an additional 10 removes, the statistics on Server A will report 20 removes and the stats on Server B will report 10 removes.

Dynamic cache events

The Dynamic Cache API allows users to register event listeners. When you are using eXtreme Scale as the dynamic cache provider, the event listeners work as expected for local in-memory caches.

For distributed caches, event behavior will depend on the topology being used. For caches using the embedded topology, events will be generated on the server that handles the write operations, also known as the primary shard. This means that only one server will receive event notifications, but it will have all the event notifications normally expected from the dynamic cache provider. Because WebSphere eXtreme Scale chooses the primary shard at runtime, it is not possible to ensure that a particular server process always receives these events.

Embedded partitioned caches will generate events on any server that hosts a partition of the cache. So if a cache has 11 partitions and each server in an 11 server WebSphere Application Server Network Deployment grid hosts one of the partitions, then each server will receive the dynamic cache events for the cache entries that it hosts. No single server process would see all of the events unless all 11 partitions were hosted in that server process. As with the embedded topology, it is not possible to ensure that a particular server process will receive a particular set of events or any events at all.

Caches that use the remote topology do not support dynamic cache events.

MBean calls

The WebSphere eXtreme Scale dynamic cache provider does not support disk caching. Any MBean calls relating to disk caching will not work.

Dynamic cache replication policy mapping

The WebSphere Application Server built-in dynamic cache provider supports multiple cache replication policies. These policies can be configured globally or on each cache entry. See the dynamic cache documentation for a description of these replication policies.

The eXtreme Scale dynamic cache provider does not honor these policies directly. The replication characteristics of a cache are determined by the configured eXtreme Scale distributed topology type and apply to all values placed in that cache, regardless of the replication policy set on the entry by the dynamic cache service. The following is a list of all the replication policies supported by the dynamic cache service and illustrates which eXtreme Scale topology provides similar replication characteristics.

Note that the eXtreme Scale dynamic cache provider ignores DRS replication policy settings on a cache or cache entry. Users must choose the topology that appropriate to their replication needs.

- NOT_SHARED – currently none of the topologies provided by the eXtreme Scale dynamic cache provider can approximate this policy. This means that all data stored into the cache must have keys and values that implement java.io.Serializable.
- SHARED_PUSH – The embedded topology approximates this replication policy. When a cache entry is created it is replicated to all the servers. Servers only look for cache entries locally. If an entry is not found locally, it is assumed to be non-existent and the other servers are not queried for it.
- SHARED_PULL and SHARED_PUSH_PULL – The embedded partitioned and remote topologies approximate this replication policy. The distributed state of the cache is completely consistent between all the servers.

This information is provided mainly so you can make sure that the topology meets your distributed consistency needs. For example, if the embedded topology is a better choice for a your deployment and performance needs, but you require the level of cache consistency provided by SHARED_PUSH_PULL, then consider using embedded partitioned, even though the performance may be slightly lower.

Security

You can secure dynamic cache instances that are running in embedded or embedded partitioned topologies with the security functionality built into WebSphere Application Server. See the documentation on Securing application servers in the WebSphere Application Server Information Center.

When a cache is running in remote topology, it is possible for a standalone eXtreme Scale client to connect to the cache and affect the contents of the dynamic cache instance. The eXtreme Scale dynamic cache provider has a low overhead encryption feature that can prevent cache data from being read or changed by non-WebSphere Application Server clients. To enable this feature, set the optional parameter **com.ibm.websphere.xs.dynacache.encryption_password** to the same value on every WebSphere Application Server instance that accesses the dynamic cache provider. This will encrypt the value and user metadata for the CacheEntry using 128-bit AES encryption. It is very important that the same value be set on all servers. Servers will not be able to read data put into the cache by servers with a different value for this parameter.

If the eXtreme Scale provider detects that different values are set for this variable in the same cache, it generate a warning in the log of the eXtreme Scale container process.

See the eXtreme Scale documentation on WebSphere eXtreme Scale security if SSL or client authentication is required.

Additional information

- Dynamic cache Redbook
- Dynamic cache documentation
  - WebSphere Application Server 7.0
  - WebSphere Application Server 6.1
- DRS documentation
  - WebSphere Application Server 7.0
  - WebSphere Application Server 6.1

Chapter 4. Scalability overview

WebSphere eXtreme Scale is scalable through the use of partitioned data, and can scale to thousands of containers if required because each container is independent from other containers.

WebSphere eXtreme Scale divides data sets into distinct partitions that can be moved between processes or even between physical servers at run time. You can, for example, start with a deployment of four servers and then expand to a deployment with 10 servers as the demands on the cache grow. Just as you can add more physical servers and processing units for vertical scalability, you can extend the elastic scaling capability horizontally with partitioning. Horizontal scaling is a major advantage to using WebSphere eXtreme Scale over an in-memory database. In-memory databases can only scale vertically.

With WebSphere eXtreme Scale, you can also use a set of APIs to gain transactional access this partitioned and distributed data. The choices you make for interacting with the cache are as significant as the functions to manage the cache for availability from a performance perspective.

Note: Scalability is not available when containers communicate with one another. The availability management, or core grouping, protocol is an $O(N^2)$ heartbeat and view maintenance algorithm, but is mitigated by keeping the number of core group members under 20. Only peer to peer replication between shards exists.

Distributed clients

The WebSphere eXtreme Scale client protocol supports very large numbers of clients. The partitioning strategy offers assistance by assuming that all clients are not always interested in all partitions because connections can be spread across multiple containers. Clients are connected directly to the partitions so latency is limited to one transferred connection.

Data grids, partitions, and shards

A data grid is divided into partitions. A partition holds an exclusive subset of the data. A partition contains one or more shards: a primary shard and replica shards. Replica shards are not necessary in a partition, but you can use replica shards to provide high availability. Whether your deployment is an independent in-memory data grid or an in-memory database processing space, data access in eXtreme Scale relies heavily on shards.

The data for a partition is stored in a set of shards at run time. This set of shards includes a primary shared and possibly one or more replica shards. A shard is the smallest unit that eXtreme Scale can add or remove from a Java virtual machine.

Two placement strategies exist: fixed partition placement (default) and per container placement. The following discussion focuses on the usage of the fixed partition placement strategy.\

Total number of shards

If your environment includes 10 partitions that hold one million objects with no replicas, then 10 shards would exist that each store 100,000 objects. If you add a replica to this scenario, then an extra shard exists in each partition. In this case, 20 shards exist: 10 primary shards and 10 replica shards. Each one of these shards store 100,000 objects. Each partition consists of a primary shard and one or more (N) replica shards. Determining the optimal shard count is critical. If you configure few shards, data is not distributed evenly among the shards, resulting in out of memory errors and processor overloading issues. You must have at least 10 shards for each JVM as you scale. When you are initially deploying the data grid, you would potentially use many partitions.

Number of shards per JVM scenarios

Scenario: small number of shards for each JVM

Data is added and removed from a JVM using shard units. Shards are never split into pieces. If 10 GB of data existed, and 20 shards exist to hold this data, then each shard holds 500 MB of data on average. If nine Java virtual machines host the data grid, then on average each JVM has two shards. Because 20 is not evenly divisible by 9, a few Java virtual machines have three shards, in the following distribution:

- Seven Java virtual machines with two shards
- Two Java virtual machines with three shards

Because each shard holds 500 MB of data, the distribution of data is unequal. The seven Java virtual machines with two shards each host 1 GB of data. The two Java virtual machines with three shards have 50% more data, or 1.5 GB, which is a much larger memory burden. Because the two Java virtual machines are hosting three shards, they also receive 50% more requests for their data. As a result, having few shards for each JVM causes imbalance. To increase the performance, you increase the number of shards for each JVM.

Scenario: increased number of shards per JVM

In this scenario, consider a much larger number of shards. In this scenario, there are 101 shards with nine Java virtual machines hosting 10 GB of data. In this case, each shard holds 99 MB of data. The Java virtual machines have the following distribution of shards:

- Seven Java virtual machines with 11 shards
- Two Java virtual machines with 12 shards

The two Java virtual machines with 12 shards now have just 99 MB more data than the other shards, which is a 9% difference. This scenario is much more evenly distributed than the 50% difference in the scenario with few shards. From a processor use perspective, only 9% more work exists for the two Java virtual machines with the 12 shards compared to the seven Java virtual machines that have 11 shards. By increasing the number of shards in each JVM, the data and processor use is distributed in a fair and even way.

When you are creating your system, use 10 shards for each JVM in its maximally sized scenario, or when the system is running its maximum number of Java virtual machines in your planning horizon.

Additional placement factors

The number of partitions, the placement strategy, and number and type of replicas are set in the deployment policy. The number of shards that are placed depend on the deployment policy that you define. The numInitialContainers, minSyncReplicas, developmentMode, maxSyncReplicas, and maxAsyncReplicas attributes affect where and when partitions and replicas can be placed. If the maximum number of replicas are not placed during the initial startup, additional replicas might be placed if you start additional servers later. When you are planning the number of shards per JVM, the maximum number of primary and replica shards is dependent on having enough JVMs started to support the configured maximum number of replicas. A replica is never placed in the same process as its primary. If a process is lost, both the primary and the replica are lost. When the developmentMode attribute is set to `false`, the primary and replicas are not placed on the same phyiscal server.

Partitioning

Use partitioning to scale out an application. You can define the number of partitions in your deployment policy.

About partitioning

Partitioning is not like Redundant Array of Independent Disks (RAID) striping, which slices each instance across all stripes. Each partition hosts the complete data for individual entries. Partitioning is a very effective means for scaling, but is not applicable to all applications. Applications that require transactional guarantees across large sets of data do not scale and cannot be partitioned effectively. WebSphere eXtreme Scale does not currently support two-phase commit across partitions.

Important: Select the number of partitions carefully. The number of partitions that are defined in the deployment policy directly affects the number of container servers to which an application can scale. Each partition is made up of a primary shard and the configured number of replica shards. The `(Number_Partitions*(1 + Number_Replicas))` formula is the number of containers that can be used to scale out a single application.

Using partitions

A data grid can have up to thousands of partitions. A data grid can scale up to the product of the number of partitions times the number of shards per partition. For example, if you have 16 partitions and each partition has one primary and one replica, or two shards, then you can potentially scale to 32 Java virtual machines. In this case, one shard is defined for each JVM. You must choose a reasonable number of partitions based on the expected number of Java virtual machines that you are likely to use. Each shard increases processor and memory usage for the system. The system is designed to scale out to handle this overhead in line with how many server Java virtual machines are available.

Applications should not use thousands of partitions if the application runs on a data grid of four container server Java virtual machines. The application should be configured to have a reasonable number of shards for each container server JVM. For example, an unreasonable configuration is 2000 partitions with two shards that are running on four container Java virtual machines. This configuration results in 4000 shards that are placed on four container Java virtual machines or 1000 shards per container JVM.

A better configuration would be under 10 shards for each expected container JVM. This configuration still gives the possibility of allowing for elastic scaling that is ten times the initial configuration while keeping a reasonable number of shards per container JVM.

Consider this scaling example: you currently have six physical servers with two container Java virtual machines per physical server. You expect to grow to 20 physical servers over the next three years. With 20 physical servers, you have 40 container server Java virtual machines, and choose 60 to be pessimistic. You want four shards per container JVM. You have 60 potential containers, or a total of 240 shards. If you have a primary and replica per partition, then you want 120 partitions. This example gives you 240 divided by 12 container Java virtual machines, or 20 shards per container JVM for the initial deployment with the potential to scale out to 20 computers later.

ObjectMap and partitioning

With the default FIXED_PARTITION placement strategy, maps are split across partitions and keys hash to different partitions. The client does not need to know to which partition the keys belong. If a mapSet has multiple maps, the maps should be committed in separate transactions.

Entities and partitioning

Entity manager entities have an optimization that helps clients that are working with entities on a server. The entity schema on the server for the map set can specify a single root entity. The client must access all entities through the root entity. The entity manager can then find related entities from that root in the same partition without requiring the related maps to have a common key. The root entity establishes affinity with a single partition. This partition is used for all entity fetches within the transaction after affinity is established. This affinity can save memory because the related maps do not require a common key. The root entity must be specified with a modified entity annotation as shown in the following example:

```
@Entity(schemaRoot=true)
```

Use the entity to find the root of the object graph. The object graph defines the relationships between one or more entities. Each linked entity must resolve to the same partition. All child entities are assumed to be in the same partition as the root. The child entities in the object graph are only accessible from a client from the root entity. Root entities are always required in partitioned environments when using an eXtreme Scale client to communicate to the server. Only one root entity type can be defined per client. Root entities are not required when using Extreme Transaction Processing (XTP) style ObjectGrids, because all communication to the partition is accomplished through direct, local access and not through the client and server mechanism.

Placement and partitions

You have two placement strategies available for WebSphere eXtreme Scale: fixed partition and per-container. The choice of placement strategy affects how your deployment configuration places partitions over the remote data grid.

Fixed partition placement

You can set the placement strategy in the deployment policy XML file. The default placement strategy is fixed-partition placement, enabled with the `FIXED_PARTITION` setting. The number of primary shards that are placed across the available containers is equal to the number of partitions that you have configured with the numberOfPartitions attribute. If you have configured replicas, the minimum total number of shards placed is defined by the following formula: `((1 primary shard + minimum synchronous shards) * partitions defined)`. The maximum total number of shards placed is defined by the following formula: `((1 primary shard + maximum synchronous shards + maximum asynchronous shards) * partitions)`. Your WebSphere eXtreme Scale deployment spreads these shards over the available containers. The keys of each map are hashed into assigned partitions based on the total partitions you have defined. They keys hash to the same partition even if the partition moves because of failover or server changes.

For example, if the numberPartitions value is 6 and the minSync value is 1 for MapSet1, the total shards for that map set is 12 because each of the 6 partitions requires a synchronous replica. If three containers are started, WebSphere eXtreme Scale places four shards per container for MapSet1.

Per-container placement

The alternate placement strategy is per-container placement, which is enabled with the `PER_CONTAINER` setting for the placementStrategy attribute in the map set element in the deployment XML file. With this strategy, the number of primary shards placed on each new container is equal to the number of partitions, *P*, that you have configured. The WebSphere eXtreme Scale deployment environment places *P* replicas of each partition for each remaining container. The numInitialContainers setting is ignored when you are using per-container placement. The partitions get larger as the containers grow. The keys for maps are not fixed to a certain partition in this strategy. The client routes to a partition and uses a random primary. If a client wants to reconnect to the same session that it used to find a key again, it must use a session handle.

For more information, see the topic on using a SessionHandle for routing in the *Programming Guide*.

For failover or stopped servers, the WebSphere eXtreme Scale environment moves the primary shards in the per-container placement strategy if they still contain data. If the shards are empty, they are discarded. In the per-container strategy, old primary shards are not kept because new primary shards are placed for every container.

WebSphere eXtreme Scale allows per-container placement as an alternative to what could be termed the "typical" placement strategy, a fixed-partition approach with the key of a Map hashed to one of those partitions. In a per-container case (which you set with PER_CONTAINER), your deployment places the partitions on the set of online container servers and automatically scales them out or in as containers are added or removed from the server data grid. A data grid with the fixed-partition approach works well for key-based grids, where the application uses a key object to locate data in the grid. The following discusses the alternative.

Example of a per-container data grid

PER_CONTAINER data grids are different. You specify that the data grid uses the PER_CONTAINER placement strategy with the placementStrategy attribute in your deployment XML file. Instead of configuring how many partitions total you want in the data grid, you specify how many partitions you want per container that you start.

For example, if you set five partitions per container, five new anonymous partition primaries are created when you start that container server, and the necessary replicas are created on the other deployed container servers.

The following is a potential sequence in a per-container environment as the data grid grows.

1. Start container C0 hosting 5 primaries (P0 - P4).
   - C0 hosts: P0, P1, P2, P3, P4.
2. Start container C1 hosting 5 more primaries (P5 - P9). Replicas are balanced on the containers.
   - C0 hosts: P0, P1, P2, P3, P4, R5, R6, R7, R8, R9.
   - C1 hosts: P5, P6, P7, P8, P9, R0, R1, R2, R3, R4.
3. Start container C2 hosting 5 more primaries (P10 - P14). Replicas are balanced further.
   - C0 hosts: P0, P1, P2, P3, P4, R7, R8, R9, R10, R11, R12.
   - C1 hosts: P5, P6, P7, P8, P9, R2, R3, R4, R13, R14.
   - C2 hosts: P10, P11, P12, P13, P14, R5, R6, R0, R1.

The pattern continues as more containers are started, creating five new primary partitions each time and rebalancing replicas on the available containers in the data grid.

Note: WebSphere eXtreme Scale does not move primary shards when using the PER_CONTAINER strategy, only replicas.

Remember that the partition numbers are arbitrary and have nothing to do with keys, so you cannot use key-based routing. If a container stops then the partition IDs created for that container are no longer used, so there is a gap in the partition IDs. In the example, there would no longer be partitions P5 - P9 if the container C2 failed, leaving only P0 - P4 and P10 - P14, so key-based hashing is impossible.

Using numbers like five or even more likely 10 for how many partitions per container works best if you consider the consequences of a container failure. To spread the load of hosting shards evenly across the data grid, you need more than just one partition for each container. If you had a single partition per container, then when a container fails, only one container (the one hosting the corresponding replica shard) must bear the full load of the lost primary. In this case, the load is immediately doubled for the container. However, if you have five partitions per container, then five containers pick up the load of the lost container, lowering impact on each by 80 percent. Using multiple partitions per container generally lowers the potential impact on each container substantially. More directly, consider a case in which a container spikes unexpectedly–the replication load of that container is spread over 5 containers rather than only one.

Using the per-container policy

Several scenarios make the per-container strategy an ideal configuration, such as with HTTP session replication or application session state. In such a case, an HTTP router assigns a session to a servlet container. The servlet container needs to create an HTTP session and chooses one of the 5 local partition primaries for the session. The "ID" of the partition chosen is then stored in a cookie. The servlet container now has local access to the session state which means zero latency access to the data for this request as long as you maintain session affinity. And eXtreme Scale replicates any changes to the partition.

In practice, remember the repercussions of a case in which you have multiple partitions per container (say 5 again). Of course, with each new container started, you have 5 more partition primaries and 5 more replicas. Over time, more partitions should be created and they should not move or be destroyed. But this is not how the containers would actually behave. When a container starts, it hosts 5 primary shards, which can be called "home" primaries, existing on the respective containers that created them. If the container fails, the replicas become primaries and eXtreme Scale creates 5 more replicas to maintain high availability (unless you disabled auto repair). The new primaries are in a different container than the one that created them, which can be called "foreign" primaries. The application should never place new state or sessions in a foreign primary. Eventually, the foreign primary has no entries and eXtreme Scale automatically deletes it and its associated replicas. The foreign primaries' purpose is to allow existing sessions to still be available (but not new sessions).

A client can still interact with a data grid that does not rely on keys. The client just begins a transaction and stores data in the data grid independent of any keys. It asks the Session for a SessionHandle object, a serializable handle allowing the client to interact with the same partition when necessary. For more information see the topic on using a SessionHandle for routing in the *Programming Guide*. WebSphere eXtreme Scale chooses a partition for the client from the list of home partition primaries. It does not return a foreign primary partition. The SessionHandle can be serialized in an HTTP cookie, for example, and later convert the cookie back into a SessionHandle. Then the WebSphere eXtreme Scale APIs can obtain a Session bound to the same partition again, using the SessionHandle.

Note: You cannot use agents to interact with a PER_CONTAINER data grid.

Advantages

The previous description is different from a normal FIXED_PARTITION or hash data grid because the per-container client stores data in a place in the grid, gets a handle to it and uses the handle to access it again. There is no application-supplied key as there is in the fixed-partition case.

Your deployment does not make a new partition for each Session. So in a per-container deployment, the keys used to store data in the partition must be unique within that partition. For example, you may have your client generate a unique SessionID and then use it as the key to find information in Maps in that partition. Multiple client sessions then interact with the same partition so the application needs to use unique keys to store session data in each given partition.

The previous examples used 5 partitions, but the numberOfPartitions parameter in the objectgrid XML file can be used to specify the partitions as required. Instead of per data grid, the setting is per container. (The number of replicas is specified in the same way as with the fixed-partition policy.)

The per-container policy can also be used with multiple zones. If possible, eXtreme Scale returns a SessionHandle to a partition whose primary is located in the same zone as that client. The client can specify the zone as a parameter to the container or by using an API. The client zone ID can be set using `serverproperties` or `clientproperties`.

The PER_CONTAINER strategy for a data grid suits applications storing conversational type state rather than database-oriented data. The key to access the data would be a conversation ID and is not related to a specific database record. It provides higher performance (because the partition primaries can be collocated with the servlets for example) and easier configuration (without having to calculate partitions and containers).

Single-partition and cross-data-grid transactions

The major distinction between WebSphere eXtreme Scale and traditional data storage solutions like relational databases or in-memory databases is the use of partitioning, which allows the cache to scale linearly. The important types of transactions to consider are single-partition and every-partition (cross-data-grid) transactions.

In general, interactions with the cache can be categorized as single-partition transactions or cross-data-grid transactions, as discussed in the following section.

Single-partition transactions

Single-partition transactions are the preferable method for interacting with caches that are hosted by WebSphere eXtreme Scale. When a transaction is limited to a single partition, then by default it is limited to a single Java virtual machine, and therefore a single server computer. A server can complete *M* number of these transactions per second, and if you have *N* computers, you can complete `M*N` transactions per second. If your business increases and you need to perform twice as many of these transactions per second, you can double *N* by buying more computers. Then you can meet capacity demands without changing the application, upgrading hardware, or even taking the application offline.

In addition to letting the cache scale so significantly, single-partition transactions also maximize the availability of the cache. Each transaction only depends on one computer. Any of the other `(N-1)` computers can fail without affecting the success or response time of the transaction. So if you are running 100 computers and one of them fails, only 1 percent of the transactions in flight at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale relocates the partitions that are hosted by the failed server to the other 99 computers. During this brief period, before the operation completes, the other 99 computers can still complete transactions. Only the transactions that would involve the partitions that are being relocated are blocked. After the failover process is complete, the cache can continue running, fully operational, at 99 percent of its original throughput capacity. After the failed server is replaced and returned to the data grid, the cache returns to 100 percent throughput capacity.

Cross-data-grid transactions

In terms of performance, availability and scalability, cross-data-grid transactions are the opposite of single-partition transactions. Cross-data-grid transactions access every partition and therefore every computer in the configuration. Each computer in the data grid is asked to look up some data and then return the result. The transaction cannot complete until every computer has responded, and therefore the throughput of the entire data grid is limited by the slowest computer. Adding computers does not make the slowest computer faster and therefore does not improve the throughput of the cache.

Cross-data-grid transactions have a similar effect on availability. Extending the previous example, if you are running 100 servers and one server fails, then 100 percent of the transactions that are in progress at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale starts to relocate the partitions that are hosted by that server to the other 99 computers. During this time, before the failover process completes, the data grid cannot process any of these transactions. After the failover process is complete, the cache can continue running, but at reduced capacity. If each computer in the data grid serviced 10 partitions, then 10 of the remaining 99 computers receive at least one extra partition as part of the failover process. Adding an extra partition increases the workload of that computer by at least 10 percent. Because the throughput of the data grid is limited to the throughput of the slowest computer in a cross-data-grid transaction, on average, the throughput is reduced by 10 percent.

Single-partition transactions are preferable to cross-data-grid transactions for scaling out with a distributed, highly available, object cache like WebSphere eXtreme Scale. Maximizing the performance of these kinds of systems requires the use of techniques that are different from traditional relational methodologies, but you can turn cross-data-grid transactions into scalable single-partition transactions.

Best practices for building scalable data models

The best practices for building scalable applications with products like WebSphere eXtreme Scale include two categories: foundational principles and implementation tips. Foundational principles are core ideas that need to be captured in the design of the data itself. An application that does not observe these principles is unlikely to scale well, even for its mainline transactions. Implementation tips are applied for problematic transactions in an otherwise well-designed application that observes the general principles for scalable data models.

Foundational principles

Some of the important means of optimizing scalability are basic concepts or principles to keep in mind.

Duplicate instead of normalizing

The key thing to remember about products like WebSphere eXtreme Scale is that they are designed to spread data across a large number of computers. If the goal is to make most or all transactions complete on a single partition, then the data model design needs to ensure that all the data the transaction might need is located in the partition. Most of the time, the only way to achieve this is by duplicating data.

For example, consider an application like a message board. Two very important transactions for a message board are showing all the posts from a given user and all the posts on a given topic. First consider how these transactions would work with a normalized data model that contains a user record, a topic record, and a post record that contains the actual text. If posts are partitioned with user records, then displaying the topic becomes a cross-grid transaction, and vice versa. Topics and users cannot be partitioned together because they have a many-to-many relationship.

The best way to make this message board scale is to duplicate the posts, storing one copy with the topic record and one copy with the user record. Then, displaying the posts from a user is a single-partition transaction, displaying the posts on a topic is a single-partition transaction, and updating or deleting a post is a two-partition transaction. All three of these transactions will scale linearly as the number of computers in the data grid increases.

Scalability rather than resources

The biggest obstacle to overcome when considering denormalized data models is the impact that these models have on resources. Keeping two, three, or more copies of some data can seem to use too many resources to be practical. When you are confronted with this scenario, remember the following facts: Hardware resources get cheaper every year. Second, and more importantly, WebSphere eXtreme Scale eliminates most hidden costs associated with deploying more resources.

Measure resources in terms of cost rather than computer terms such as megabytes and processors. Data stores that work with normalized relational data generally need to be located on the same computer. This required collocation means that a single larger enterprise computer needs to be purchased rather than several smaller computers. With enterprise hardware, it is not uncommon for one computer to be capable of completing one million transactions per second to cost much more than the combined cost of 10 computers capable of doing 100,000 transactions per second each.

A business cost in adding resources also exists. A growing business eventually runs out of capacity. When you run out of capacity, you either need to shut down while moving to a bigger, faster computer, or create a second production environment to which you can switch. Either way, additional costs will come in the form of lost business or maintaining almost twice the capacity needed during the transition period.

With WebSphere eXtreme Scale, the application does not need to be shut down to add capacity. If your business projects that you need 10 percent more capacity for the coming year, then increase the number of computers in the data grid by 10 percent. You can increase this percentage without application downtime and without purchasing excess capacity.

Avoid data transformations

When you are using WebSphere eXtreme Scale, data should be stored in a format that is directly consumable by the business logic. Breaking the data down into a more primitive form is costly. The transformation needs to be done when the data is written and when the data is read. With relational databases this transformation is done out of necessity, because the data is ultimately persisted to disk quite frequently, but with WebSphere eXtreme Scale, you do not need to perform these transformations. For the most part data is stored in memory and can therefore be stored in the exact form that the application needs.

Observing this simple rule helps denormalize your data in accordance with the first principle. The most common type of transformation for business data is the JOIN operations that are necessary to turn normalized data into a result set that fits the needs of the application. Storing the data in the correct format implicitly avoids performing these JOIN operations and produces a denormalized data model.

Eliminate unbounded queries

No matter how well you structure your data, unbounded queries do not scale well. For example, do not have a transaction that asks for a list of all items sorted by value. This transaction might work at first when the total number of items is 1000, but when the total number of items reaches 10 million, the transaction returns all 10 million items. If you run this transaction, the two most likely outcomes are the transaction timing out, or the client encountering an out-of-memory error.

The best option is to alter the business logic so that only the top 10 or 20 items can be returned. This logic alteration keeps the size of the transaction manageable no matter how many items are in the cache.

Define schema

The main advantage of normalizing data is that the database system can take care of data consistency behind the scenes. When data is denormalized for scalability, this automatic data consistency management no longer exists. You must implement a data model that can work in the application layer or as a plug-in to the distributed data grid to guarantee data consistency.

Consider the message board example. If a transaction removes a post from a topic, then the duplicate post on the user record needs to be removed. Without a data model, it is possible a developer would write the application code to remove the post from the topic and forget to remove the post from the user record. However, if the developer were using a data model instead of interacting with the cache directly, the removePost method on the data model could pull the user ID from the post, look up the user record, and remove the duplicate post behind the scenes.

Alternately, you can implement a listener that runs on the actual partition that detects the change to the topic and automatically adjusts the user record. A listener might be beneficial because the adjustment to the user record could happen locally if the partition happens to have the user record, or even if the user record is on a different partition, the transaction takes place between servers instead of between the client and server. The network connection between servers is likely to be faster than the network connection between the client and the server.

Avoid contention

Avoid scenarios such as having a global counter. The data grid will not scale if a single record is being used a disproportionate number of times compared to the rest of the records. The performance of the data grid will be limited by the performance of the computer that holds the given record.

In these situations, try to break the record up so it is managed per partition. For example consider a transaction that returns the total number of entries in the distributed cache. Instead of having every insert and remove operation access a single record that increments, have a listener on each partition track the insert and remove operations. With this listener tracking, insert and remove can become single-partition operations.

Reading the counter will become a cross-data-grid operation, but for the most part, it was already as inefficient as a cross-data-grid operation because its performance was tied to the performance of the computer hosting the record.

Implementation tips

You can also consider the following tips to achieve the best scalability.

Use reverse-lookup indexes

Consider a properly denormalized data model where customer records are partitioned based on the customer ID number. This partitioning method is the logical choice because nearly every business operation performed with the customer record uses the customer ID number. However, an important transaction that does not use the customer ID number is the login transaction. It is more common to have user names or e-mail addresses for login instead of customer ID numbers.

The simple approach to the login scenario is to use a cross-data-grid transaction to find the customer record. As explained previously, this approach does not scale.

The next option might be to partition on user name or e-mail. This option is not practical because all the customer ID based operations become cross-data-grid transactions. Also, the customers on your site might want to change their user name or e-mail address. Products like WebSphere eXtreme Scale need the value that is used to partition the data to remain constant.

The correct solution is to use a reverse lookup index. With WebSphere eXtreme Scale, a cache can be created in the same distributed grid as the cache that holds all the user records. This cache is highly available, partitioned and scalable. This cache can be used to map a user name or e-mail address to a customer ID. This cache turns login into a two partition operation instead of a cross-grid operation. This scenario is not as good as a single-partition transaction, but the throughput still scales linearly as the number of computers increases.

Compute at write time

Commonly calculated values like averages or totals can be expensive to produce because these operations usually require reading a large number of entries. Because reads are more common than writes in most applications, it is efficient to compute these values at write time and then store the result in the cache. This practice makes read operations both faster and more scalable.

Optional fields

Consider a user record that holds a business, home, and telephone number. A user could have all, none or any combination of these numbers defined. If the data were normalized then a user table and a telephone number table would exist. The telephone numbers for a given user could be found using a JOIN operation between the two tables.

De-normalizing this record does not require data duplication, because most users do not share telephone numbers. Instead, empty slots in the user record must be allowed. Instead of having a telephone number table, add three attributes to each user record, one for each telephone number type.

This addition of attributes eliminates the JOIN operation and makes a telephone number lookup for a user a single-partition operation.

Placement of many-to-many relationships

Consider an application that tracks products and the stores in which the products are sold. A single product is sold in many stores, and a single store sells many products. Assume that this application tracks 50 large retailers. Each product is sold in a maximum of 50 stores, with each store selling thousands of products.

Keep a list of stores inside the product entity (arrangement A), instead of keeping a list of products inside each store entity (arrangement B). Looking at some of the transactions this application would have to perform illustrates why arrangement A is more scalable.

First look at updates. With arrangement A, removing a product from the inventory of a store locks the product entity. If the data grid holds 10000 products, only 1/10000 of the grid needs to be locked to perform the update. With arrangement B, the data grid only contains 50 stores, so 1/50 of the grid must be locked to complete the update. So even though both of these could be considered single-partition operations, arrangement A scales out more efficiently.

Now, considering reads with arrangement A, looking up the stores at which a product is sold is a single-partition transaction that scales and is fast because the transaction only transmits a small amount of data. With arrangement B, this transaction becomes an cross-data-grid transaction because each store entity must be accessed to see if the product is sold at that store, which reveals an enormous performance advantage for arrangement A.

Scaling with normalized data

One legitimate use of cross-data-grid transactions is to scale data processing. If a data grid has 5 computers and a cross-data-grid transaction is dispatched that sorts through about 100,000 records on each computer, then that transaction sorts through 500,000 records. If the slowest computer in the data grid can perform 10 of these transactions per second, then the data grid is capable of sorting through 5,000,000 records per second. If the data in the grid doubles, then each computer must sort through 200,000 records, and each transaction sorts through 1,000,000 records. This data increase decreases the throughput of the slowest computer to 5 transactions per second, thereby reducing the throughput of the data grid to 5 transactions per second. Still, the data grid sorts through 5,000,000 records per second.

In this scenario, doubling the number of computer allows each computer to return to its previous load of sorting through 100,000 records, allowing the slowest computer to process 10 of these transactions per second. The throughput of the data grid stays the same at 10 requests per second, but now each transaction processes 1,000,000 records, so the grid has doubled its capacity in terms of processing records to 10,000,000 per second.

Applications such as a search engine that need to scale both in terms of data processing to accommodate the increasing size of the Internet and throughput to accommodate growth in the number of users, you must create multiple data grids, with a round robin of the requests between the grids. If you need to scale up the throughput, add computers and add another data grid to service requests. If data processing needs to be scaled up, add more computers and keep the number of data grids constant.

Scaling in units or pods

Although you can deploy a data grid over thousands of Java virtual machines, you might consider splitting the data grid into units or pods to increase the reliability and ease of testing of your configuration. A pod is a group of servers that is running the same set of applications.

Deploying a large single data grid

Testing has verified that eXtreme Scale can scale out to over 1000 JVMs. Such testing encourages building applications to deploy single data grids on large numbers of boxes. Although it is possible to do this, it is not recommended, for several reasons:

1. **Budget concerns:** Your environment cannot realistically test a 1000-server data grid. However, it can test a much smaller data grid considering budget reasons, so you do not need to buy twice the hardware, especially for such a large number of servers.
2. **Different application versions:** Requiring a large number of boxes for each testing thread is not practical. The risk is that you are not testing the same factors as you would in a production environment.
3. **Data loss:** Running a database on a single hard drive is unreliable. Any problem with the hard drive causes you to lose data. Running a growing application on a single data grid is similar. You will likely have bugs in your environment and in your applications. So placing all of the data on a single large system will often lead to a loss of large amounts of data.

Splitting the data grid

Splitting the application data grid into pods (units) is a more reliable option. A pod is a group of servers that are running a homogenous application stack. Pods can be of any size, but ideally they should consist of about 20 physical servers. Instead of having 500 physical servers in a single data grid, you can have 25 pods of 20 physical servers. A single version of an application stack should run on a given pod, but different pods can have their own versions of an application stack.

Generally, an application stack considers levels of the following components.

- Operating system
- Hardware
- JVM
- WebSphere eXtreme Scale version
- Application
- Other necessary components

A pod is a conveniently sized deployment unit for testing. Instead of having hundreds of servers for testing, it is more practical to have 20 servers. In this case, you are still testing the same configuration as you would have in production. Production uses grids with a maximum size of 20 servers, constituting a pod. You can stress-test a single pod and determine its capacity, number of users, amount of data, and transaction throughput. This makes planning easier and follows the standard of having predictable scaling at predictable cost.

Setting up a pod-based environment

In different cases, the pod does not necessarily have to have 20 servers. The purpose of the pod size is for practical testing. The size of a pod should be small enough that if a pod encounters problems in production, the fraction of transactions affected is tolerable.

Ideally, any bug impacts a single pod. A bug would only have an impact on four percent of the application transactions rather than 100 percent. In addition, upgrades are easier because they can be rolled out one pod at a time. As a result, if an upgrade to a pod creates problems, the user can switch that pod back to the prior level. Upgrades include any changes to the application, the application stack, or system updates. As much as possible, upgrades should only change a single element of the stack at a time to make problem diagnosis more precise.

To implement an environment with pods, you need a routing layer above the pods that is forwards and backwards compatible if pods get software upgrades. Also, you should create a directory that includes information about which pod has what data. You can use another eXtreme Scale data grid for this with a database behind it, preferably using the write-behind scenario.) This yields a two-tier solution. Tier 1 is the directory and is used to locate which pod handles a specific transaction. Tier 2 is composed of the pods themselves. When tier 1 identifies a pod, the setup routes each transaction to the correct server in the pod, which is usually the server holding the partition for the data used by the transaction. Optionally, you can also use a near cache on tier 1 to lower the impact associated with looking up the correct pod.

Using pods is slightly more complex than having a single data grid, but the operational, testing, and reliability improvements make it a crucial part of scalability testing.

Chapter 5. Availability overview

High availability

With high availability, WebSphere eXtreme Scale provides reliable data redundancy and detection of failures.

WebSphere eXtreme Scale self-organizes data grids of Java virtual machines into a loosely federated tree. The catalog service at the root and core groups holding containers are at the leaves of the tree. See “Caching architecture: Maps, containers, clients, and catalogs” on page 11 for more information.

Each core group is automatically created by the catalog service into groups of about 20 servers. The core group members provide health monitoring for other members of the group. Also, each core group elects a member to be the leader for communicating group information to the catalog service. Limiting the core group size allows for good health monitoring and a highly scalable environment.

Note: In a WebSphere Application Server environment, in which core group size can be altered, eXtreme Scale does not support more than 50 members per core group.

Heart beating

1. Sockets are kept open between Java virtual machines, and if a socket closes unexpectedly, this unexpected closure is detected as a failure of the peer Java virtual machine. This detection catches failure cases such as the Java virtual machine exiting very quickly. Such detection also allows recovery from these types of failures typically in less than a second.
2. Other types of failures include: an operating system panic, physical server failure, or network failure. These failures are discovered through heart beating.

Heartbeats are sent periodically between pairs of processes: When a fixed number of heartbeats are missed, a failure is assumed. This approach detects failures in N*M seconds. N is the number of missed heart beats and M is the heartbeat interval. Directly specifying M and N is not supported. A slider mechanism is used to allow a range of tested M and N combinations to be used.

Failures

There are several ways that a process can fail. The process could fail because some resource limit was reached, such as maximum heap size, or some process control logic terminated a process. The operating system could fail, causing all of the processes running on the system to be lost. Hardware can fail, though less frequently, like the network interface card (NIC), causing the operating system to be disconnected from the network. Many more points of failure can occur, causing the process to be unavailable. In this context, all of these failures can be categorized into one of two types: process failure and loss of connectivity.

Process failure

WebSphere eXtreme Scale reacts to process failures quickly. When a process fails, the operating system is responsible for cleaning up any left over resources that the process was using. This cleanup includes port allocation and connectivity. When a process fails, a signal is sent over the connections that were being used by that process to close each connection. With these signals, a process failure can be instantly detected by any other process that is connected to the failed process.

Loss of connectivity

Loss of connectivity occurs when the operating system becomes disconnected. As a result, the operating system cannot send signals to other processes. There are several reasons that loss of connectivity can occur, but they can be split into two categories: host failure and islanding.

Host failure

If the machine is unplugged from the power outlet, then it is gone instantly.

Islanding

This scenario presents the most complicated failure condition for software to handle correctly because the process is presumed to be unavailable, though it is not. Essentially, a server or other process appears to the system to have failed while it is actually running properly.

Container failures

Container failures are generally discovered by peer containers through the core group mechanism. When a container or set of containers fails, the catalog service migrates the shards that were hosted on that container or containers. The catalog service looks for a synchronous replica first before migrating to an asynchronous replica. After the primary shards are migrated to new host containers, the catalog service looks for new host containers for the replicas that are now missing.

Note: Container islanding - The catalog service migrates shards off containers when the container is discovered to be unavailable. If those containers then become available, the catalog service considers the containers eligible for placement just like in the normal startup flow.

Container failure detection latency

Failures can be categorized into soft and hard failures. Soft failures are typically caused when a process fails. Such failures are detected by the operating system, which can recover used resources, such as network sockets, quickly. Typical failure detection for soft failures is less than one second. Hard failures might take up to 200 seconds to detect with the default heart beat tuning. Such failures include: physical machine crashes, network cable disconnects, or operating system failures. The run time relies on heart beating to detect hard failures which can be configured.

Catalog service failure

Because the catalog service grid is an eXtreme Scale grid, it also uses the core grouping mechanism in the same way as the container failure process. The primary difference is that the catalog service domain uses a peer election process for defining the primary shard instead of the catalog service algorithm that is used for the containers.

The placement service and the core grouping service are One of N services. A One of N service runs in one member of the high availability group. The location service and administration run in all of the members of the high availability group. The placement service and core grouping service are singletons because they are responsible for laying out the system. The location service and administration are read-only services and exist everywhere to provide scalability.

The catalog service uses replication to make itself fault tolerant. If a catalog service process fails, then the service restarts to restore the system to the wanted level of availability. If all of the processes that are hosting the catalog service fail, the data grid has a loss of critical data. This failure results in a required restart of all the container servers. Because the catalog service can run on many processes, this failure is an unlikely event. However, if you are running all of the processes on a single box, within a single blade chassis, or from a single network switch, a failure is more likely to occur. Try to remove common failure modes from boxes that are hosting the catalog service to reduce the possibility of failure.

Multiple container failures

A replica is never placed in the same process as its primary because if the process is lost, it would result in a loss of both the primary and the replica. In a development environment on a single machine, you might want to have two containers and replicate between them. You can define the development mode attribute in the deployment policy to configure a replica to be placed on the same machine as a primary. However, in production, using a single machine is not sufficient because loss of that host results in the loss of both container servers. To change between development mode on a single machine and a production mode with multiple machines, disable development mode in the deployment policy configuration file.

Table 9. Failure discovery and recovery summary

Loss type	Discovery (detection) mechanism	Recovery method
Process loss	I/O	Restart
Server loss	Heartbeat	Restart
Network outage	Heartbeat	Reestablish network and connection
Server-side hang	Heartbeat	Stop and restart server
Server busy	Heartbeat	Wait until server is available

Replication for availability

Replication provides fault tolerance and increases performance for a distributed eXtreme Scale topology.

Replication is enabled by associating BackingMaps with a MapSet.

A MapSet is a collection of maps that are categorized by partition-key. This partition-key is derived from the individual map's key by taking its hash modulo the number of partitions. Thus, if one group of maps within the MapSet has partition-key X, those maps will be stored in a corresponding partition X in the grid; if another group has partition-key Y, all of the maps will be stored in partition Y, and so on. Also, the data within the maps is replicated based on the policy defined on the MapSet, which is only used for distributed eXtreme Scale topologies (unnecessary for local instances).

See "Partitioning" on page 83 for more details.

MapSets are assigned what number of partitions they will have and a replication policy. The MapSet replication configuration simply identifies the number of synchronous and asynchronous replica shards a MapSet should have in addition to the primary shard. For example, if there is to be 1 synchronous and 1 asynchronous replica, all of the BackingMaps assigned to the MapSet will each have a replica shard distributed automatically within the set of available containers for the eXtreme Scale. The replication configuration can also enable clients to read data from synchronously replicated servers. This can spread the load for read requests over additional servers in the eXtreme Scale. Replication only has a programming model impact when preloading the BackingMaps.

For details on the various configuration options, see below:

Map preloading

Maps can be associated with Loaders. A loader is used to fetch objects when they cannot be found in the map (a cache miss) and also to write changes to a back-end when a transaction commits. Loaders can also be used for preloading data into a map. The preloadMap method of the Loader interface is called on each map when its corresponding partition in the MapSet becomes a primary. The preloadMap method is not called on replicas. It attempts to load all the intended referenced data from the back-end into the map using the provided session. The relevant map is identified by the BackingMap argument that is passed to the preloadMap method.

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

Preloading in partitioned MapSet

Maps can be partitioned into N partitions. Maps can therefore be striped across multiple servers, with each entry identified by a key that is stored only on one of those servers. Very large maps can be held in an eXtreme Scale because the application is no longer limited by the heap size of a single JVM to hold all the entries of a Map. Applications that want to preload with the preloadMap method of the Loader interface must identify the subset of the data that it preloads. A fixed number of partitions always exists. You can determine this number by using the following code example:

```
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();
int myPartition = backingMap.getPartitionId();
```

This code example shows how an application can identify the subset of the data to preload from the database. Applications must always use these methods even when the map is not initially partitioned. These methods allow flexibility: If the map is later partitioned by the administrators, then the loader continues to work correctly.

The application must issue queries to retrieve the *myPartition* subset from the backend. If a database is used, then it might be easier to have a column with the partition identifier for a given record unless there is some natural query that allows the data in the table to partition easily.

See details on writing a loader with a replica preload controller in the *Programming Guide* for an example on how to implement a Loader for a replicated eXtreme Scale.

Performance

The preload implementation copies data from the back-end into the map by storing multiple objects in the map in a single transaction. The optimal number of records to store per transaction depends on several factors, including complexity and size. For example, after the transaction includes blocks of more than 100 entries, the performance benefit decreases as you increase the number of entries. To determine the optimal number, begin with 100 entries and then increase the number until the performance benefit decreases to none. Larger transactions result in better replication performance. Remember, only the primary runs the preload code. The preloaded data is replicated from the primary to any replicas that are online.

Preloading MapSets

If the application uses a MapSet with multiple maps then each map has its own loader. Each loader has a preload method. Each map is loaded serially by the eXtreme Scale. It might be more efficient to preload all the maps by designating a single map as the preloading map. This process is an application convention. For example, two maps, department and employee, might use the department Loader to preload both the department and the employee maps. This procedure ensures that, transactionally, if an application wants a department then the employees for that department are in the cache. When the department Loader preloads a department from the back-end, it also fetches the employees for that department. The department object and its associated employee objects are then added to the map using a single transaction.

Recoverable preloading

Some customers have very large data sets that need caching. Preloading this data can be very time consuming. Sometimes, the preloading must complete before the application can go online. You can benefit from making preloading recoverable. Suppose there are a million records to preload. The primary is preloading them and fails at the 800,000th record. Normally, the replica chosen to be the new primary clears any replicated state and starts from the beginning. eXtreme Scale can use a ReplicaPreloadController interface. The loader for the application would also need to implement the ReplicaPreloadController interface. This example adds a single method to the Loader: `Status checkPreloadStatus(Session session, BackingMap bmap);`. This method is called by the eXtreme Scale run time before the preload method of the Loader interface is normally called. The eXtreme Scale tests the result of this method (Status) to determine its behavior whenever a replica is promoted to a primary.

Table 10. Status value and response

Returned status value	eXtreme Scale response
Status.PRELOADED_ALREADY	eXtreme Scale does not call the preload method at all because this status value indicates that the map is fully preloaded.
Status.FULL_PRELOAD_NEEDED	eXtreme Scale clears the map and calls the preload method normally.
Status.PARTIAL_PRELOAD_NEEDED	eXtreme Scale leaves the map as-is and calls preload. This strategy allows the application loader to continue preloading from that point onwards.

Clearly, while a primary is preloading the map, it must leave some state in a map in the MapSet that is being replicated so that the replica determines what status to return. You can use an extra map named, for example, RecoveryMap. This

RecoveryMap must be part of the same MapSet that is being preloaded to ensure that the map is replicated consistently with the data being preloaded. A suggested implementation follows.

As the preload commits each block of records, the process also updates a counter or value in the RecoveryMap as part of that transaction. The preloaded data and the RecoveryMap data are replicated atomically to the replicas. When the replica is promoted to primary, it can now check the RecoveryMap to see what has happened.

The RecoveryMap can hold a single entry with the state key. If no object exists for this key then you need a full preload (checkPreloadStatus returns FULL_PRELOAD_NEEDED). If an object exists for this state key and the value is COMPLETE, the preload completes, and the checkPreloadStatus method returns PRELOADED_ALREADY. Otherwise, the value object indicates where the preload restarts and the checkPreloadStatus method returns PARTIAL_PRELOAD_NEEDED. The loader can store the recovery point in an instance variable for the loader so that when preload is called, the loader knows the starting point. The RecoveryMap can also hold an entry per map if each map is preloaded independently.

Handling recovery in synchronous replication mode with a Loader

The eXtreme Scale run time is designed not to lose committed data when the primary fails. The following section shows the algorithms used. These algorithms apply only when a replication group uses synchronous replication. A loader is optional.

The eXtreme Scale run time can be configured to replicate all changes from a primary to the replicas synchronously. When a synchronous replica is placed, it receives a copy of the existing data on the primary shard. During this time, the primary continues to receives transactions and copies them to the replica asynchronously. The replica is not considered to be online at this time.

After the replica catches up the primary, the replica enters peer mode and synchronous replication begins. Every transaction committed on the primary is sent to the synchronous replicas and the primary waits for a response from each replica. A synchronous commit sequence with a Loader on the primary looks like the following set of steps:

Table 11. Commit sequence on the primary

Step with loader	Step without loader
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same
Send changes to replicas and wait for acknowledgement	same
Commit to the loader through the TransactionCallback plug-in	plug-in commit called, but does nothing
Release locks for entries	same

Notice that the changes are sent to the replica before they are committed to the loader. To determine when the changes are committed on the replica, revise this sequence: At initialize time, initialize the tx lists on the primary as below.

```
CommitedTx = {}, RolledBackTx = {}
```

During synchronous commit processing, use the following sequence:

Table 12. Synchronous commit processing

Step with loader	Step without loader
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same
Send changes with a committed transaction, roll back transaction to replica, and wait for acknowledgement	same
Clear list of committed transactions and rolled back transactions	same
Commit the loader through the TransactionCallBack plug-in	TransactionCallBack plug-in commit is still called, but typically does not do anything
If commit succeeds, add the transaction to the committed transactions, otherwise add to the rolled back transactions	no-op
Release locks for entries	same

For replica processing, use the following sequence:

1. Receive changes
2. Commit all received transactions in the committed transaction list
3. Roll back all received transactions in the rolled back transaction list
4. Start a transaction or session
5. Apply changes to the transaction or session
6. Save the transaction or session to the pending list
7. Send back reply

Notice that on the replica, no loader interactions occur while the replica is in replica mode. The primary must push all changes through the Loader. The replica does not make any changes. A side effect of this algorithm is that the replica always has the transactions, but they are not committed until the next primary transaction sends the commit status of those transactions. The transactions are then committed or rolled back on the replica. Until then, the transactions are not committed. You can add a timer on the primary that sends the transaction outcome after a small period of time (a few seconds). This timer limits, but does not eliminate, any staleness to that time window. This staleness is only a problem when using replica read mode. Otherwise, the staleness does not have an impact on the application.

When the primary fails, it is likely that a few transactions were committed or rolled back on the primary, but the message never made it to the replica with these outcomes. When a replica is promoted to the new primary, one of the first actions is to handle this condition. Each pending transaction is reprocessed against the new primary's set of maps. If there is a Loader, then each transaction is given to the Loader. These transactions are applied in strict first in first out (FIFO) order. If a transactions fails, it is ignored. If three transactions are pending, A, B, and C, then A might commit, B might rollback and C might also commit. No one transaction has any impact on the others. Assume that they are independent.

A loader might want to use slightly different logic when it is in failover recovery mode versus normal mode. The loader can easily know when it is in failover recovery mode by implementing the ReplicaPreloadController interface. The checkPreloadStatus method is only called when failover recovery completes. Therefore, if the apply method of the Loader interface is called before the checkPreloadStatus method, then it is a recovery transaction. After the checkPreloadStatus method is called, the failover recovery is complete.

Load balancing across replicas

The eXtreme Scale, unless configured otherwise, sends all read and write requests to the primary server for a given replication group. The primary must service all requests from clients. You might want to allow read requests to be sent to replicas of the primary. Sending read requests to the replicas allows the load of the read requests to be shared by multiple Java Virtual Machines (JVM). However, using replicas for read requests can result in inconsistent responses.

Load balancing across replicas is typically used only when clients are caching data that is changing all the time or when the clients are using pessimistic locking.

If the data is continually changing and then being invalidated in client near caches, the primary should see a relatively high get request rate from clients as a result. Likewise, in pessimistic locking mode, no local cache exists, so all requests are sent to the primary.

If the data is relatively static or if pessimistic mode is not used, then sending read requests to the replica does not have a big impact on performance. The frequency of get requests from clients with caches that are full of data is not high.

When a client first starts, its near cache is empty. Cache requests to the empty cache are forwarded to the primary. The client cache gets data over time, causing the request load to drop. If a large number of clients start concurrently, then the load might be significant and replica read might be an appropriate performance choice.

Client-side replication

With eXtreme Scale, you can replicate a server map to one or more clients by using asynchronous replication. A client can request a local read-only copy of a server side map by using the ClientReplicableMap.enableClientReplication method.

```
void enableClientReplication(Mode mode, int[] partitions,
ReplicationMapListener listener) throws ObjectGridException;
```

The first parameter is the replication mode. This mode can be a continuous replication or a snapshot replication. The second parameter is an array of partition IDs that represent the partitions from which to replicate the data. If the value is null or an empty array, the data is replicated from all the partitions. The last parameter is a listener to receive client replication events. See ClientReplicableMap and ReplicationMapListener in the API documentation for details.

After the replication is enabled, then the server starts to replicate the map to the client. The client is eventually only a few transactions behind the server at any point in time.

High-availability catalog service

A catalog service domain is the data grid of catalog servers you are using, which retain topology information for all of the containers in your eXtreme Scale environment. The catalog service controls balancing and routing for all clients. To deploy eXtreme Scale as an in-memory database processing space, you must cluster the catalog service into a catalog service domain for high availability.

Components of the catalog service domain

When multiple catalog servers start, one of the servers is elected as the master catalog server that accepts Internet Inter-ORB Protocol (IIOP) heartbeats and handles system data changes in response to any catalog service or container changes.

When clients contact any one of the catalog servers, the routing table for the catalog service domain is propagated to the clients through the Common Object Request Broker Architecture (CORBA) service context.

Configure at least three catalog servers. Catalog servers must be installed on separate nodes or separate installation images from your container servers to ensure that you can seamlessly upgrade your servers at a later date. If your configuration has zones, you can configure one catalog server per zone.

When an eXtreme Scale server and container contacts one of the catalog servers, the routing table for the catalog service domain is also propagated to the eXtreme Scale server and container through the CORBA service context. Furthermore, if the contacted catalog server is not currently the master catalog server, the request is automatically rerouted to the current master catalog server and the routing table for the catalog server is updated.

Note: A catalog service domain and the container server data grid are very different. The catalog service domain is for high availability of your system data. The container server data grid is for your data high availability, scalability, and workload management. Therefore, two different routing tables exist: the routing table for the catalog service domain and the routing table for the container server data grid shards.

The catalog service domain responsibilities are divided into a series of services:

Core group manager

The catalog service uses the high availability manager (HA manager) to group processes together for availability monitoring. Each grouping of the processes is a core group. With eXtreme Scale, the core group manager dynamically groups the processes together. These processes are kept small to allow for scalability. Each core group elects a leader that has the added responsibility of sending status to the core group manager when individual members fail. The same status mechanism is used to discover when all the members of a group fail, which causes the communication with the leader to fail.

The core group manager is a fully automatic service responsible for organizing containers into small groups of servers that are then automatically loosely federated to make an ObjectGrid. When a container first contacts the catalog service, it waits to be assigned to either a new or existing group. An eXtreme Scale deployment consists of many such groups, and this grouping is a key scalability enabler. Each group is a group of Java virtual machines that uses heart beating to monitor the availability of the other groups. One of these group members is elected the leader and has an added responsibility to relay availability information to the catalog service to allow for failure reaction by reallocation and route forwarding.

Placement service
The catalog service manages the placement of shards across the set of available container servers. The placement service is responsible for maintaining balance across physical resources. The placement service is responsible for allocating individual shards to their host container. The placement service runs as a One of N elected service in the data grid, so exactly one instance of the service is running. If that instance fails, another process is then elected and it takes over. For redundancy, the state of the catalog service is replicated across all the servers that are hosting the catalog service.

Administration
The catalog service is also the logical entry point for system administration. The catalog service hosts an Managed Bean (MBean) and provides Java Management Extensions (JMX) URLs for any of the servers that the catalog service is managing.

Location service
The location service acts as the touchpoint for both clients that are searching for the containers that host the application they seek, as well as for the container servers that are registering hosted applications with the placement service. The location service runs on all of the data grid members to scale out this function.

Catalog service domain deployment

The catalog service hosts logic that is typically idle during steady states. As a result, the catalog service minimally influences scalability. The service is built to service hundreds of containers that become available simultaneously. For availability, configure the catalog service into a data grid.

Planning

After a catalog service domain is started, the members of the data grid bind together. Carefully plan your catalog service domain topology, because you cannot modify your catalog service domain configuration at run time. Spread out the data grid as diversely as possible to prevent errors.

Starting a catalog service domain

For more information about creating a catalog service domain, see the details about starting a catalog service domain in the *Administration Guide*.

Connecting eXtreme Scale containers embedded in WebSphere Application Server to a stand-alone catalog service domain

You can configure eXtreme Scale containers that are embedded in a WebSphere Application Server environment to connect to a stand-alone catalog service domain.

- 7.1+ You can create catalog service domains in the WebSphere Application Server administrative console. See the details about creating catalog service domains in the *Administration Guide* for more information.
- (deprecated) In previous releases, you connected the catalog services into a catalog service domain by creating a custom property. This property can still be used, but is deprecated. For more information about this custom property, see the information about starting the catalog service process in a WebSphere Application Server in the *Administration Guide*..

  Note: Server name collision: Because this property is used to start the eXtreme Scale catalog server as well as to connect to it, catalog servers must not have the same name as any WebSphere Application Server server.

See "Catalog server quorums" for more information.

Catalog server quorums

When the quorum mechanism is enabled, all the catalog servers in the quorum must be available for placement operations to occur in the data grid.

- "Important terms"
- "Heartbeats and failure detection"
- "Quorum behavior" on page 108
  - "Container behavior during quorum loss" on page 111
- "Client behavior during quorum loss" on page 111

Important terms

- **Heartbeat**: A signal that is sent between servers to convey that they are running.
- **Quorum**: A group of catalog servers that communicate and conduct placement operations in the data grid. This group consists of all of the catalog servers in the data grid, unless you manually override the quorum mechanism with administrative actions.
- **Brownout**: A temporary loss of connectivity between one or more servers.
- **Blackout**: A permanent loss of connectivity between one or more servers.
- **Data center**: A geographically located group of servers that are generally connected with a local area network (LAN).
- **Zone**: A zone is a configuration option that is used to group servers together that share some physical characteristic. Examples of zones for a group of servers include: a data center, an area network, a building, or a floor of a building.
- **Heartbeat**: Heartbeats are used to determine if a given Java virtual machine (JVM) is running.

Heartbeats and failure detection

Container servers and core groups

The catalog service places container servers into core groups of a limited size. A core group tries to detect the failure of its members. A single member of a core group is elected to be the core group leader. The core group leader periodically tells the catalog service that the core group is alive and reports any membership changes to the catalog service. A membership change can be a JVM failing or a newly added JVM that joins the core group.

If a JVM socket is closed, that JVM is regarded as being no longer available. Each core group member also heart beats over these sockets at a rate determined by configuration. If a JVM does not respond to these heartbeats within a configured maximum time period, then the JVM is considered to be no longer available, which triggers a failure detection.

If the catalog service marks a container JVM as failed and the container server is later reported as being available, the container JVM is told to shut down the WebSphere eXtreme Scale container servers. A JVM in this state is not visible in **xsadmin** command queries. Messages in the logs of the container JVM indicate that the container JVM has failed. You must manually restart these JVMs.

If the core group leader cannot contact any member, it continues to retry contacting the member.

The complete failure of all members of a core group is also a possibility. If the entire core group has failed, it is the responsibility of the catalog service to detect this loss.

Catalog service domain heart-beating

The catalog service domain looks like a private core group with a static membership and a quorum mechanism. It detects failures the same way as a normal core group. However, the behavior is modified to include quorum logic. The catalog service also uses a less aggressive heart-beating configuration.

Failure detection

WebSphere eXtreme Scale detects when processes terminate through abnormal socket closure events. The catalog service is notified immediately when a process terminates.

For more information about configuring heart-beating, see the information about configuring failover detection in the *Administration Guide*.

Quorum behavior

Normally, the members of the catalog service have full connectivity. The catalog service domain is a static set of JVMs. WebSphere eXtreme Scale expects all members of the catalog service to be online. When all the members are online, the catalog service has quorum. The catalog service responds to container events only while the catalog service has quorum.

Reasons for quorum loss

WebSphere eXtreme Scale expects to lose quorum for the following scenarios:
- A catalog service JVM member fails
- Network brown out occurs
- Data center loss occurs

WebSphere eXtreme Scale does not lose quorum in the following scenario:

- Stopping a catalog server instance with the **stopOgServer** command or any other administrative actions. The system knows that the server instance has stopped, which is different from a JVM failure or brownout.

If the catalog service loses a quorum, it waits for quorum to be reestablished. While the catalog service does not have a quorum, it ignores events from container servers. Container servers continue to try any requests that are rejected by the catalog server during this time. Heart-beating is suspended until a quorum is reestablished.

Quorum loss from JVM failure

A catalog server that fails causes quorum to be lost. If a JVM fails, you must override quorum as fast as possible. The failed catalog service cannot rejoin the data grid until quorum has been overridden.

Quorum loss from network brownout

WebSphere eXtreme Scale is designed to expect the possibility of brownouts. A brownout is when a temporary loss of connectivity occurs between data centers. Brown outs are usually transient and clear within seconds or minutes. While WebSphere eXtreme Scale tries to maintain normal operation during the brownout period, a brownout is regarded as a single failure event. The failure is expected to be fixed and then normal operation resumes with no actions necessary.

A long duration brown out can be classified as a blackout only through user intervention. Overriding quorum on one side of the brownout is required in order for the event to be classified as a blackout.

Catalog service JVM cycling

If a catalog server is stopped by using the **stopOgServer** command, then the quorum drops to one less server. The remaining servers still have quorum. Restarting the catalog server sets quorum back to the previous number.

Consequences of lost quorum

If a container JVM was to fail while quorum is lost, recovery does not occur until the brownout recovers. In a blackout scenario, the recovery does not occur until you run the override quorum command. Quorum loss and a container failure as are considered a double failure, which is a rare event. Because of the double failure, applications might lose write access to data that was stored on the failed JVM. When quorum is restored, the normal recovery occurs.

Similarly, if you attempt to start a container during a quorum loss event, the container does not start.

Full client connectivity is allowed during quorum loss. If no container failures or connectivity issues happen during the quorum loss event then clients can still fully interact with the container servers.

If a brownout occurs, then some clients might not have access to primary or replica copies of the data until the brownout clears.

New clients can be started because a catalog service JVM must exist in each data center. Therefore, at least one catalog server can be reached by a client even during a brownout event.

Quorum recovery

If quorum is lost for any reason, when quorum is reestablished, a recovery protocol is run. When the quorum loss event occurs, all liveness checking for core groups is suspended and failure reports are also ignored. After quorum is back, then the catalog service checks all the core groups to immediately determine their membership. Any shards previously hosted on container JVMs reported as failed are recovered. If primary shards were lost, then surviving replicas are promoted to being primary shards. If replica shards were lost then additional replicas shards are created on the survivors.

Overriding quorum

Override quorum only when a data center failure has occurred. Quorum loss due to a catalog service JVM failure or a network brownout recovers automatically after the catalog service JVM is restarted or the network brownout ends.

Administrators are the only ones with knowledge of a data center failure. WebSphere eXtreme Scale treats a brownout and a blackout similarly. You must inform the WebSphere eXtreme Scale environment of such failures with the **`xsadmin`** command to override quorum. This command tells the catalog service to assume that quorum is achieved with the current membership, and full recovery takes place. When issuing an override quorum command, you are guaranteeing that the JVMs in the failed data center have truly failed and do not have a chance of recovering.

The following list considers some scenarios for overriding quorum. In this scenario, you have three catalog servers: A, B, and C.

- **Brown out:** The C catalog server is isolated temporarily. The catalog service loses quorum and waits for the brownout to complete. After the brownout is over, the C catalog server rejoins the catalog service domain and quorum is reestablished. Your application sees no problems during this time.
- **Temporary failure:** During a temporary failure, the C catalog server fails and the catalog service loses quorum. You must override quorum. After quorum is reestablished, you can restart the C catalog server. The C catalog server joins the catalog service domain again when it restarts. Your application sees no problems during this time.
- **Data center failure:** You verify that the data center has failed and that it has been isolated on the network. Then you issue the **`xsadmin`** override quorum command. The surviving two data centers run a full recovery by replacing shards that were hosted in the failed data center. The catalog service is now running with a full quorum of the A and B catalog servers. The application might see delays or exceptions during the interval between the start of the blackout and when quorum is overridden. After quorum is overridden, the data grid recovers and normal operation is resumed.
- **Data center recovery:** The surviving data centers are already running with quorum overridden. When the data center that contains the C catalog server is restarted, all JVMs in the data center must be restarted. Then the C catalog server joins the existing catalog service domain again and the quorum setting reverts to the normal situation with no user intervention.

- **Data center failure and brownout:** The data center that contains the C catalog server fails. Quorum is overridden and recovered on the remaining data centers. If a brownout between the A and B catalog servers occurs, the normal brownout recovery rules apply. After the brownout clears, quorum is reestablished and necessary recovery from the quorum loss occurs.

Container behavior during quorum loss

Containers host one or more shards. Shards are either primaries or replicas for a specific partition. The catalog service assigns shards to a container and the container server uses that assignment until new instructions arrive from the catalog service. For example, a primary shard continues to try communication with its replica shards during network brownouts, until the catalog service provides further instructions to the primary shard.

Synchronous replica behavior

The primary shard can accept new transactions while the connection is broken if the number of replicas online are at least at the `minsync` property value for the map set. If any new transactions are processed on the primary shard while the link to the synchronous replica is broken, the replica is and resynchronized with the current state of the primary when the link is reestablished.

Do not configure synchronous replication between data centers or over a WAN-style link.

Asynchronous replica behavior

While the connection is broken, the primary shard can accept new transactions. The primary shard buffers the changes up to a limit. If the connection with the replica is reestablished before that limit is reached then the replica is updated with the buffered changes. If the limit is reached, then the primary destroys the buffered list and when the replica reattaches then it is cleared and resynchronized.

Client behavior during quorum loss

Clients are always able to connect to the catalog server to bootstrap to the data grid whether the catalog service domain has quorum or not. The client tries to connect to any catalog server instance to obtain a route table and then interact with the data grid. Network connectivity might prevent the client from interacting with some partitions due to network setup. The client might connect to local replicas for remote data if it has been configured to do so. Clients cannot update data if the primary partition for that data is not available.

Replicas and shards

With eXtreme Scale, an in-memory database or shard can be replicated from one Java virtual machine (JVM) to another. A shard represents a partition that is placed on a container. Multiple shards that represent different partitions can exist on a single container. Each partition has an instance that is a primary shard and a configurable number of replica shards. The replica shards are either synchronous or asynchronous. The types and placement of replica shards are determined by eXtreme Scale using a deployment policy, which specifies the minimum and maximum number of synchronous and asynchronous shards.

Shard types

Replication uses three types of shards:

- Primary
- Synchronous replica
- Asynchronous replica

The primary shard receives all insert, update and remove operations. The primary shard adds and removes replicas, replicates data to the replicas, and manages commits and rollbacks of transactions.

Synchronous replicas maintain the same state as the primary. When a primary replicates data to a synchronous replica, the transaction is not committed until it commits on the synchronous replica.

Asynchronous replicas might or might not be at the same state as the primary. When a primary replicates data to an asynchronous replica, the primary does not wait for the asynchronous replica to commit.

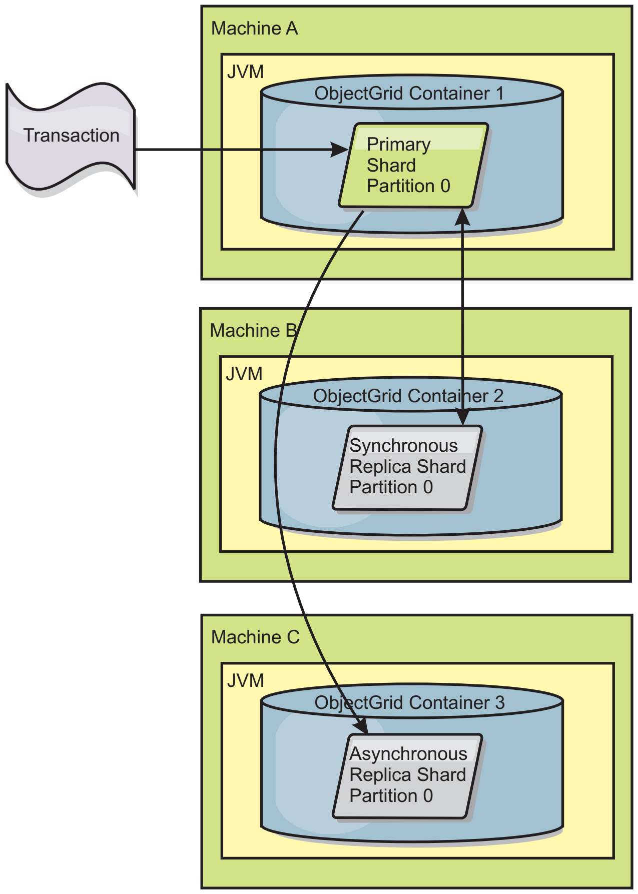


Figure 34. Communication path between a primary shard and replica shards

Minimum synchronous replica shards

When a primary prepares to commit data, it checks how many synchronous replica shards voted to commit the transaction. If the transaction processes normally on the replica, it votes to commit. If something went wrong on the synchronous replica, it votes not to commit. Before a primary commits, the number of synchronous replica shards that are voting to commit must meet the minSyncReplica setting from the deployment policy. When the number of synchronous replica shards that are voting to commit is too low, the primary does not commit the transaction and an error results. This action ensures that the required number of synchronous replicas are available with the correct data. Synchronous replicas that encountered errors reregister to fix their state. For more information about reregistering, see Replica shard recovery.

The primary throws a ReplicationVotedToRollbackTransactionException error if too few synchronous replicas voted to commit.

Replication and Loaders

Normally, a primary shard writes changes synchronously through the Loader to a database. The Loader and database are always in sync. When the primary fails over to a replica shard, the database and Loader might not be in synch. For example:

- The primary can send the transaction to the replica and then fail before committing to the database.
- The primary can commit to the database and then fail before sending to the replica.

Either approach leads to either the replica being one transaction in front of or behind the database. This situation is not acceptable. eXtreme Scale uses a special protocol and a contract with the Loader implementation to solve this issue without two phase commit. The protocol follows:

Primary side

- Send the transaction along with the previous transaction outcomes.
- Write to the database and try to commit the transaction.
- If the database commits, then commit on eXtreme Scale. If the database does not commit, then roll back the transaction.
- Record the outcome.

Replica side

- Receive a transaction and buffer it.
- For all outcomes, send with the transaction, commit any buffered transactions and discard any rolled back transactions.

Replica side on failover

- For all buffered transactions, provide the transactions to the Loader and the Loader attempts to commit the transactions.
- The Loader needs to be written to make each transaction is idempotent.
- If the transaction is already in the database, then the Loader performs no operation.
- If the transaction is not in the database, then the Loader applies the transaction.
- After all transactions are processed, then the new primary can begin to serve requests.

This protocol ensures that the database is at the same level as the new primary state.

Shard placement

The catalog service is responsible for placing shards. Each ObjectGrid has a number of partitions, each of which has a primary shard and an optional set of replica shards. The catalog service does not place replica and primary shards for the same partition on the same container. It also does not place replica and primary shards on containers that have the same IP address (unless the configuration is in development mode). The catalog service allocates the shards by balancing them so that they are evenly distributed over the available containers.

If a new container starts, then eXtreme Scale retrieves shards from relatively overloaded containers to the new empty container. With this behavior, eXtreme Scale establishes and maintains its essential elasticity. The elasticity is manifest in its powerful ability for scaling horizontally, both to scale out and scale in.

Scaling out

Scaling out means that when extra Java virtual machines, or containers, are added to an eXtreme Scale data grid, then eXtreme Scale tries to move existing shards, primaries or replicas, from the old set of JVMs to the new set. This movement allows the data grid to expand to take advantage of the processor, network and memory of the newly added JVMs. The movement also balances the data grid and tries to ensure that each JVM in the grid hosts the same amount of data. As the data grid expands, each server hosts a smaller subset of the total grid. eXtreme Scale assumes that data is distributed evenly among the partitions. This expansion enables scaling out.

Scaling in

Scaling in means that if a JVM fails, then eXtreme Scale tries to repair the damage. If the failed JVM had a replica, then eXtreme Scale replaces the lost replica by creating a new replica on a surviving JVM. If the failed JVM had a primary, then eXtreme Scale finds the best replica on the survivors and promotes the replica to be the new primary. eXtreme Scale then replaces the promoted replica with a new replica that is created on the remaining servers. To maintain scalability, eXtreme Scale preserves the replica count for partitions as servers fail.

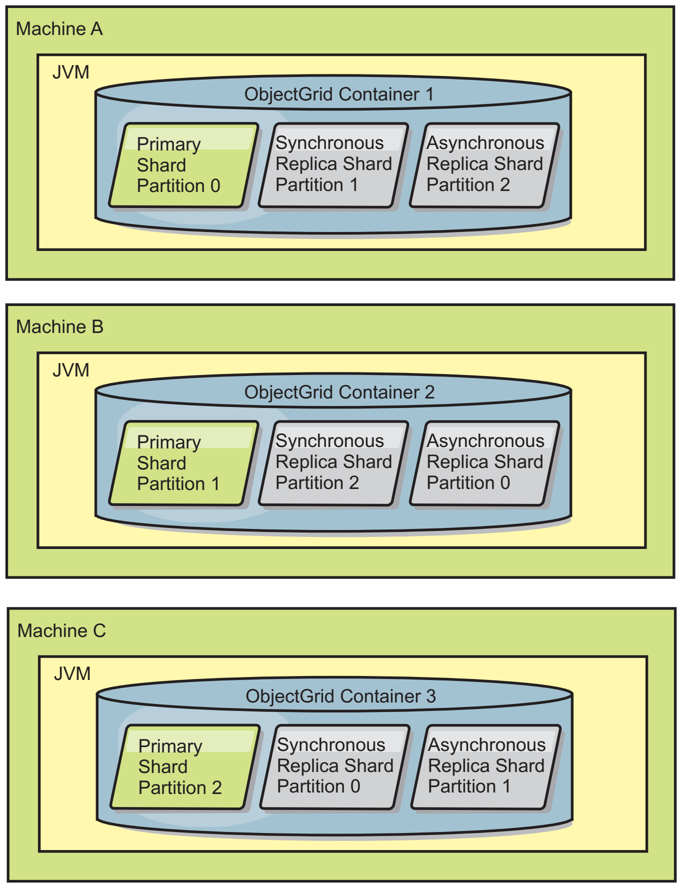

Machine A
JVM
ObjectGrid Container 1
Primary Shard Partition 0
Synchronous Replica Shard Partition 1
Asynchronous Replica Shard Partition 2
Machine B
JVM
ObjectGrid Container 2
Primary Shard Partition 1
Synchronous Replica Shard Partition 2
Asynchronous Replica Shard Partition 0
Machine C
JVM
ObjectGrid Container 3
Primary Shard Partition 2
Synchronous Replica Shard Partition 0
Asynchronous Replica Shard Partition 1

Figure 35. Placement of an ObjectGrid mapset with a deployment policy of 3 partitions with a minSyncReplicas value of 1, a maxSyncReplicas value of 1, and a maxAsyncReplicas value of 1

Reading from replicas

You can configure map sets such that a client is permitted to read from a replica rather than being restricted to primary shards only.

It can often be advantageous to allow replicas to serve as more than simply potential primaries in the case of failures. For example, map sets can be configured to allow read operations to be routed to replicas by setting the replicaReadEnabled option on the MapSet to true. The default setting is false.

For more information on the MapSet element, see the topic on the deployment policy descriptor XML file in the *Administration Guide*.

Enabling reading of replicas can improve performance by spreading read requests to more Java™ virtual machines. If the option is not enabled, all read requests such as the ObjectMap.get or the Query.getResultIterator methods are routed to the primary. When replicaReadEnabled is set to true, some get requests might return stale data, so an application using this option must be able to tolerate this possibility. However, a cache miss will not occur. If the data is not on the replica, the get request is redirected to the primary and tried again.

The replicaReadEnabled option can be used with both synchronous and asynchronous replication.

Load balancing across replicas

Load balancing across replicas is typically used only when clients are caching data that is changing all the time or when the clients are using pessimistic locking.

The eXtreme Scale, unless configured otherwise, sends all read and write requests to the primary server for a given replication group. The primary must service all requests from clients. You might want to allow read requests to be sent to replicas of the primary. Sending read requests to the replicas allows the load of the read requests to be shared by multiple Java Virtual Machines (JVM). However, using replicas for read requests can result in inconsistent responses.

Load balancing across replicas is typically used only when clients are caching data that is changing all the time or when the clients are using pessimistic locking.

If the data is continually changing and then being invalidated in client near caches, the primary should see a relatively high get request rate from clients as a result. Likewise, in pessimistic locking mode, no local cache exists, so all requests are sent to the primary.

If the data is relatively static or if pessimistic mode is not used, then sending read requests to the replica does not have a big impact on performance. The frequency of get requests from clients with caches that are full of data is not high.

When a client first starts, its near cache is empty. Cache requests to the empty cache are forwarded to the primary. The client cache gets data over time, causing the request load to drop. If a large number of clients start concurrently, then the load might be significant and replica read might be an appropriate performance choice.

Shard life cycles

Shards go through different states and events to support replication. The life cycle of a shard includes coming online, run time, shut down, fail over and error handling. Shards can be promoted from a replica shard to a primary shard to handle server state changes.

Life cycle events

When primary and replica shards are placed and started, they go through a series of events to bring themselves online and into listening mode.

Primary shard

The catalog service places a primary shard for a partition. The catalog service also does the work of balancing primary shard locations and initiating failover for primary shards.

When a shard becomes a primary shard, it receives a list of replicas from the catalog service. The new primary shard creates a replica group and registers all the replicas.

When the primary is ready, an open for business message displays in the `SystemOut.log` file for the container on which it is running. The open message, or the CWOBJ1511I message, lists the map name, map set name, and partition number of the primary shard that started.

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (primary) is open for business.
```

See “Shard placement” on page 114 for more information on how the catalog service places shards.

Replica shard

Replica shards are mainly controlled by the primary shard unless the replica shard detects a problem. During a normal life cycle, the primary shard places, registers, and de-registers a replica shard.

When the primary shard initializes a replica shard, a message displays the log that describes where the replica runs to indicate that the replica shard is available. The open message, or the CWOBJ1511I message, lists the map name, map set name, and partition number of the replica shard. This message follows:

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (synchronous replica) is open for business.
```

or

```
CWOBJ1511I: mapName:mapSetName:partitionNumber (asynchronous replica) is open for business.
```

Asynchronous replica shard: An asynchronous replica shard polls the primary for data. The replica automatically will adjust the poll timing if it does not receive data from the primary, which indicates that it is caught up with the primary. It also will adjust if it receives an error that might indicate that the primary has failed, or if there is a network problem.

When the asynchronous replica starts replicating, it prints the following message to the SystemOut.log file for the replica. This message might print more than one time per CWOBJ1511 message. It will print again if the replica connects to a different primary or if template maps are added.

```
CWOBJ1543I: The asynchronous replica objectGridName:mapsetName:partitionNumber started or
continued replicating from the primary. Replicating for maps: [mapName]
```

Synchronous replica shard: When the synchronous replica shard first starts, it is not yet in peer mode. When a replica shard is in peer mode, it receives data from the primary as data comes into the primary. Before entering peer mode, the replica shard needs a copy of all of the existing data on the primary shard.

The synchronous replica copies data from the primary shard similar to an asynchronous replica by polling for data. When it copies the existing data from the primary, it switches to peer mode and begins to receive data as the primary receives the data.

When a replica shard reaches peer mode, it prints a message to the SystemOut.log file for the replica. The time refers to the amount of time that it took the replica shard to get all of its initial data from the primary shard. The time might display as zero or very low if the primary shard does not have any existing data to replicate. This message may print more than one time per CWOBJ1511 message. It will print again if the replica connects to a different primary or if template maps are added.

```
CWOBJ1526I: Replica objectGridName:mapsetName:partitionNumber:mapName entering peer
mode after X seconds.
```

When the synchronous replica shard is in peer mode, the primary shard must replicate transactions to all peer mode synchronous replicas. The synchronous replica shard data remains at the same level as the primary shard data. If a minimum number of synchronous replicas or minSync is set in the deployment policy, that number of synchronous replicas must vote to commit before the transaction can successfully commit on the primary.

Recovery events

Replication is designed to recover from failure and error events. If a primary shard fails, another replica takes over. If errors are on the replica shards, the replica shard attempts to recover. The catalog service controls the placement and transactions of new primary shards or new replica shards.

Replica shard becomes a primary shard

A replica shard becomes a primary shard for two reasons. Either the primary shard stopped or failed, or a balance decision was made to move the previous primary shard to a new location.

The catalog service selects a new primary shard from the existing synchronous replica shards. If a primary move needs to take place and there are no replicas, a temporary replica will be placed to complete the transition. The new primary shard registers all of the existing replicas and accepts transactions as the new primary shard. If the existing replica shards have the correct level of data, the current data is preserved as the replica shards register with the new primary shard. Asynchronous replicas will poll against the new primary.

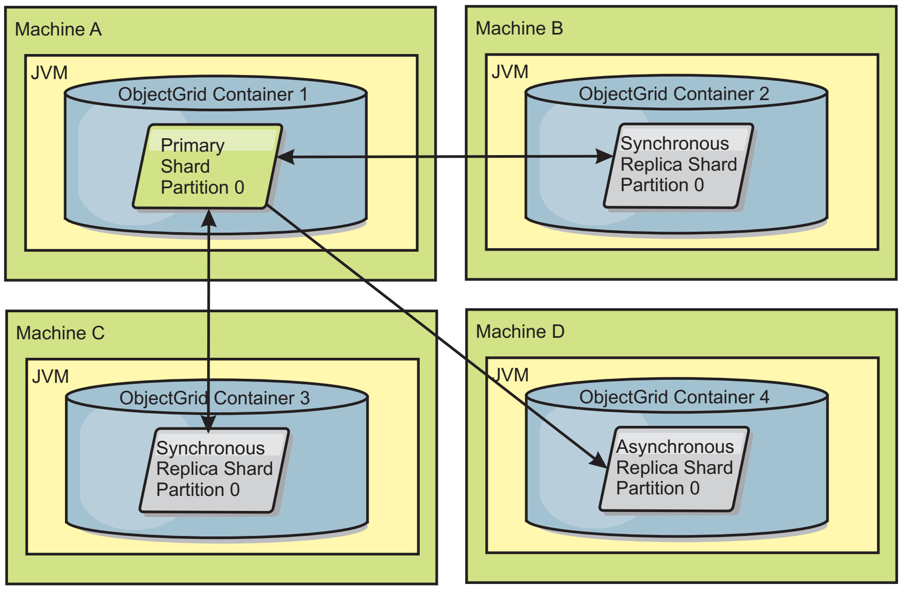

Machine A
JVM
ObjectGrid Container 1
Primary Shard Partition 0
Machine B
JVM
ObjectGrid Container 2
Synchronous Replica Shard Partition 0
Machine C
JVM
ObjectGrid Container 3
Synchronous Replica Shard Partition 0
Machine D
JVM
ObjectGrid Container 4
Asynchronous Replica Shard Partition 0

Figure 36. Example placement of an ObjectGrid map set for the partition0 partition. The deployment policy has a minSyncReplicas value of 1, a maxSyncReplicas value of 2, and a maxAsyncReplicas value of 1.

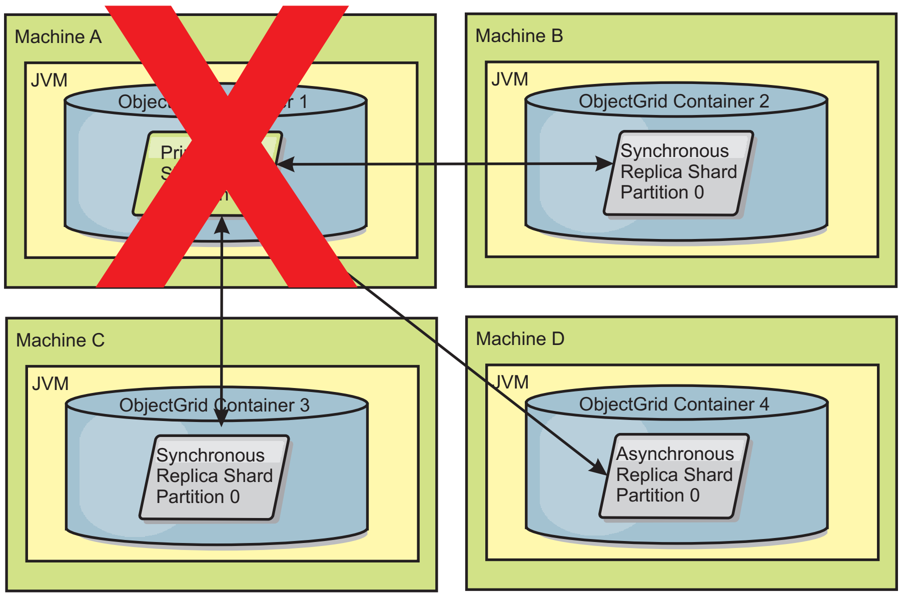


Figure 37. The container for the primary shard fails

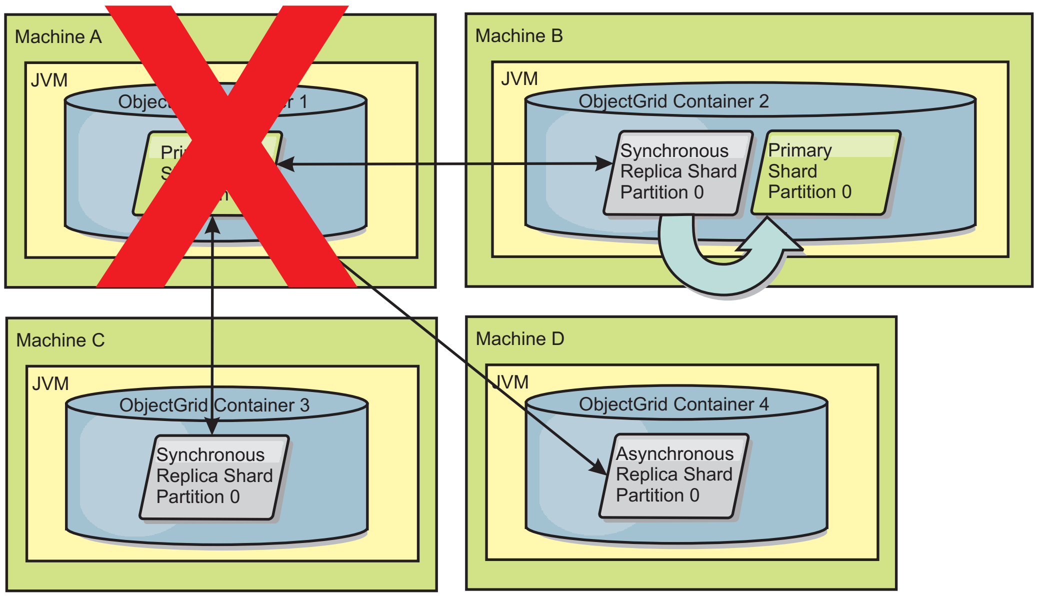


Figure 38. The synchronous replica shard on ObjectGrid container 2 becomes the primary shard

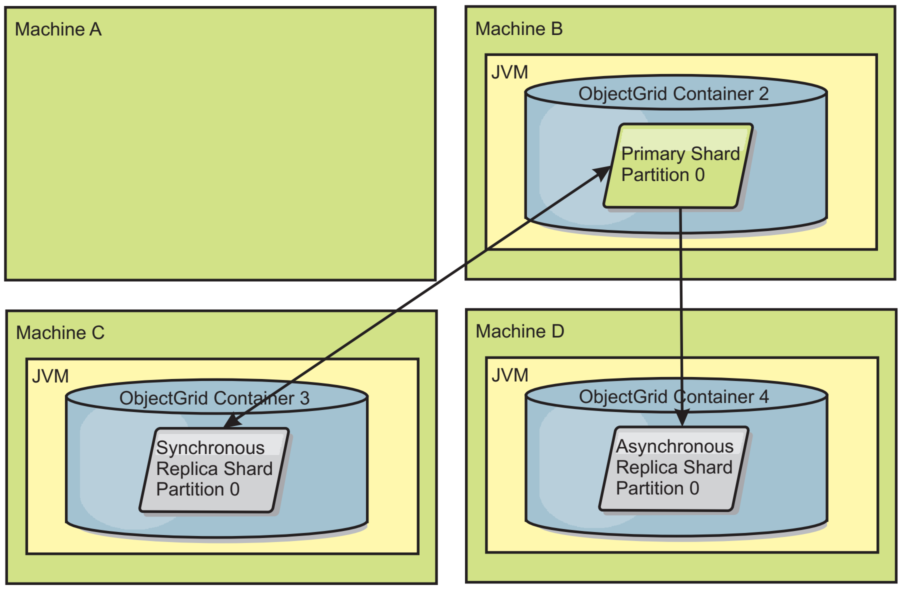


Figure 39. Machine B contains the primary shard. Depending on how automatic repair mode is set and the availability of the containers, a new synchronous replica shard might or might not be placed on a machine.

Replica shard recovery

A synchronous replica shard is controlled by the primary shard. However, if a replica shard detects a problem, it can trigger a reregister event to correct the state of the data. The replica clears the current data and gets a fresh copy from the primary.

When a replica shard initiates a reregister event, the replica prints a log message.

```
CWOBJ1524I: Replica listener
objectGridName:mapSetName:partition must re-register with the primary.
Reason: Exception listed
```

If a transaction causes an error on a replica shard during processing, then the replica shard is in an unknown state. The transaction successfully processed on the primary shard, but something went wrong on the replica. To correct this situation, the replica initiates a reregister event. With a new copy of data from the primary, the replica shard can continue. If the same problem reoccurs, the replica shard does not continuously reregister. See "Failure events" for more details.

Failure events

A replica can stop replicating data if it encounters error situations for which the replica cannot recover.

Too many register attempts

If a replica triggers a reregister multiple times without successfully committing data, the replica stops. Stopping prevents a replica from entering an endless reregister loop. By default, a replica shard tries to reregister three times in a row before stopping.

If a replica shard reregisters too many times, it prints the following message to the log.

```
CWOBJ1537E: objectGridName:mapSetName:partition exceeded the maximum number
of times to reregister (timesAllowed) without successful transactions..
```

If the replica is unable to recover by reregistering, a pervasive problem might exist with the transactions that are relative to the replica shard. A possible problem could be missing resources on the classpath if an error occurs while inflating the keys or values from the transaction.

Failure while entering peer mode

If a replica attempts to enter peer mode and experiences an error processing the bulk existing data from the primary (the checkpoint data), the replica shuts down. Shutting down prevents a replica from starting with incorrect initial data. Because it receives the same data from the primary if it reregisters, the replica does not retry.

If a replica shard fails to enter peer mode, it prints the following message to the log:

```
CWOBJ1527W Replica objectGridName:mapSetName:partition:mapName failed to enter peer mode after numSeconds seconds.
```

An additional message displays in the log that explains why the replica failed to enter peer mode.

Recovery after re-register or peer mode failure

If a replica fails to re-register or enter peer mode, the replica is in an inactive state until a new placement event occurs. A new placement event might be a new server starting or stopping. You can also start a placement event by using the triggerPlacement method on the PlacementServiceMBean Mbean.

Configuring zones for replica placement

Zone support allows sophisticated configurations for replica placement across data centers. With this capability, grids of thousands of partitions can be easily managed using a handful of optional placement rules. A data center can be different floors of a building, different buildings, or even different cities or other distinctions as configured with zone rules.

Flexibility of zones

You can place shards into zones. This function allows you to have more control over how eXtreme Scale places shards on a grid. Java virtual machines that host an eXtreme Scale server can be tagged with a zone identifier. The deployment file can now include one or more zone rules and these zone rules are associated with a shard type. The following section gives an overview of zone usage. For more information, consult the Configuring zones for replica placement topic in the *Administration Guide*.

Placement zones control of how the eXtreme Scale assigns out primaries and replicas to configure advanced topologies.

A Java virtual machine can have multiple containers but only 1 server. A container can host multiple shards from a single ObjectGrid.

This capability is useful to make sure that replicas and primaries are placed in different locations or zones for better high availability. Normally, eXtreme Scale does not place a primary and replica shard on Java virtual machines with the same IP address. This simple rule normally prevents two eXtreme Scale servers from being placed on the same physical computer. However, you might require a more flexible mechanism. For example, you might be using two blade chassis and want the primaries to be *striped* across both chassis and the replica for each primary be placed on the other chassis from the primary.

Striped primaries means that primaries are placed into each zone and the replica for each primary is located in the opposite zone. For example primary 0 would be in zoneA, and sync replica 0 would be in zoneB. Primary 1 would be in zoneB, and sync replica 1 would be in zoneA.

The chassis name would be the zone name in this case. Alternatively, you might name zones after floors in a building and use zones to make sure that primaries and replicas for the same data are on different floors. Buildings and data centers are also possible. Testing has been done across data centers using zones as a mechanism to ensure the data is adequately replicated between the data centers. If you are using the HTTP Session Manager for eXtreme Scale, you can also use zones. With this feature, you can deploy a single Web application across three data centers and ensure that HTTP sessions for users are replicated across data centers so that the sessions can be recovered even if an entire data center fails.

WebSphere eXtreme Scale is aware of the need to manage a large grid over multiple data centers. It can make sure that backups and primaries for the same partition are located in different data centers if that is required. It can put all primaries in data center 1 and all replicas in data center 2 or it can round robin primaries and replicas between both data centers. The rules are flexible so that many scenarios are possible. eXtreme Scale can also manage thousands of servers, which together with completely automatic placement with data center awareness makes such large grids affordable from an administrative point of view. Administrators can specify what they want to do simply and efficiently.

As an administrator, use placement zones to control where primary and replica shards are placed, which allows for the set up of advanced high performance and highly available topologies. You can define a zone to any logical grouping of eXtreme Scale processes, as noted above: These zones can correspond to physical workstation locations such as a data center, a floor of a data center, or a blade chassis. You can stripe data across zones, which provides increased availability, or you can split the primaries and replicas into separate zones when a hot standby is required.

Associating an eXtreme Scale server with a zone that is not using WebSphere Extended Deployment

If eXtreme Scale is used with Java Standard Edition or an application server that is not based on WebSphere Extended Deployment Version 6.1, then a JVM that is a shard container can be associated with a zone if using the following techniques.

Applications using the startOgServer script

The startOgServer script is used to start an eXtreme Scale application when it is not being embedded in an existing server. The **`-zone`** parameter is used to specify the zone to use for all containers within the server.

Specifying the zone when starting a container using APIs

The zone name for a container can be specified as described in the Embedded server API documentation in the *Programming Guide*.

Associating WebSphere Extended Deployment nodes with zones

If you are using eXtreme Scale with WebSphere Extended Deployment Java EE applications, you can leverage WebSphere Extended Deployment node groups to place servers in specific zones.

In eXtreme Scale, a JVM is only allowed to be a member of a single zone. However, WebSphere allows a node to be a part of multiple node groups. You can use this functionality of eXtreme Scale zones if you ensure that each of your nodes is in only one zone node group.

Use the following syntax to name your node group in order to declare it a zone: `ReplicationZone<UniqueSuffix>`. Servers running on a node that is part of such a node group are included in the zone specified by the node group name. The following is a description of an example topology.

First, you configure 4 nodes: node1, node2, node3, and node4, each node having 2 servers. Then you create a node group named ReplicationZoneA and a node group named ReplicationZoneB. Next, you add node1 and node2 to ReplicationZoneA and add node3 and node4 to ReplicationZoneB.

When the servers on node1 and node2 are started, they will become part of ReplicationZoneA, and likewise the servers on node3 and node4 will become part of ReplicationZoneB.

A grid-member JVM checks for zone membership at startup only. Adding a new node group or changing the membership only has an impact on newly started or restarted JVMs.

Zone rules

An eXtreme Scale partition has one primary shard and zero or more replica shards. For this example, consider the following naming convention for these shards. `P` is the primary shard, `S` is a synchronous replica and `A` is an asynchronous replica. A zone rule has three components:

- A rule name
- A list of zones
- An inclusive or exclusive flag

The zone name for a container can be specified as described in the documentation for Embedded server API. A zone rule specifies the possible set of zones in which a shard can be placed. The inclusive flag indicates that after a shard is placed in a zone from the list, then all other shards are also placed in that zone. An exclusive setting indicates that each shard for a partition is placed in a different zone in the zone list. For example, using an exclusive setting means that if there are three shards (primary, and two synchronous replicas), then the zone list must have three zones.

Each shard can be associated with one zone rule. A zone rule can be shared between two shards. When a rule is shared then the inclusive or exclusive flag extends across shards of all types sharing a single rule.

Examples

A set of examples showing various scenarios and the deployment configuration to implement the scenarios follows.

Striping primaries and replicas across zones

You have three blade chassis, and want primaries distributed across all three, with a single synchronous replica placed in a different chassis than the primary. Define each chassis as a zone with chassis names ALPHA, BETA, and GAMMA. An example deployment XML follows:

```
<?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="library">
   <mapSet name="ms1" numberOfPartitions="37" minSyncReplicas="1"
    maxSyncReplicas="1" maxAsyncReplicas="0">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="stripeZone"/>
    <shardMapping shard="S" zoneRuleRef="stripeZone"/>
    <zoneRule name ="stripeZone" exclusivePlacement="true" >
     <zone name="ALPHA" />
     <zone name="BETA" />
     <zone name="GAMMA" />
    </zoneRule>
   </zoneMetadata>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

This deployment XML contains a grid called library with a single Map called book. It uses four partitions with a single synchronous replica. The zone metadata clause shows the definition of a single zone rule and the association of zone rules with shards. The primary and synchronous shards are both associated with the zone rule "stripeZone". The zone rule has all three zones in it and it uses exclusive placement. This rule means that if the primary for partition 0 is placed in ALPHA then the replica for partition 0 will be placed in either BETA or GAMMA. Similarly, primaries for other partitions are placed in other zones and the replicas will be placed.

Asynchronous replica in a different zone than primary and synchronous replica

In this example, two buildings exist with a high latency connection between them. You want no data loss high availability for all scenarios. However, the performance impact of synchronous replication between buildings leads you to a trade off. You want a primary with synchronous replica in one building and an asynchronous replica in the other building. Normally, the failures are JVM crashes or computer failures rather than large scale issues. With this topology, you can survive normal failures with no data loss. The loss of a building is rare enough that some data loss is acceptable in that case. You can make two zones, one for each building. The deployment XML file follows:

```
<?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="library">
  <mapSet name="ms1" numberOfPartitions="13" minSyncReplicas="1"
   maxSyncReplicas="1" maxAsyncReplicas="1">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="primarySync"/>
    <shardMapping shard="S" zoneRuleRef="primarySync"/>
    <shardMapping shard="A" zoneRuleRef="aysnc"/>
    <zoneRule name ="primarySync" exclusivePlacement="false" >
      <zone name="BldA" />
      <zone name="BldB" />
    </zoneRule>
    <zoneRule name="aysnc" exclusivePlacement="true">
      <zone name="BldA" />
      <zone name="BldB" />
    </zoneRule>
   </zoneMetadata>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

The primary and synchronous replica share a primaySync zone rule with an exclusive flag setting of false. So, after the primary or sync gets placed in a zone, then the other is also placed in the same zone. The asynchronous replica uses a second zone rule with the same zones as the primarySync zone rule but it uses the **exclusivePlacement** attribute set to true. This attribute indicates that means a shard cannot be placed in a zone with another shard from the same partition. As a result, the asynchronous replica does not get placed in the same zone as the primary or synchronous replicas.

Placing all primaries in one zone and all replicas in another zone

Here, all primaries are in one specific zone and all replicas in a different zone. We will have a primary and a single asynchronous replica. All replicas will be in zone A and primaries in B.

```
<?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="library">
  <mapSet name="ms1" numberOfPartitions="13" minSyncReplicas="0"
   maxSyncReplicas="0" maxAsyncReplicas="1">
   <map ref="book" />
   <zoneMetadata>
    <shardMapping shard="P" zoneRuleRef="primaryRule"/>
    <shardMapping shard="A" zoneRuleRef="replicaRule"/>
    <zoneRule name ="primaryRule">
     <zone name="A" />
    </zoneRule>
    <zoneRule name="replicaRule">
     <zone name="B" />
      </zoneRule>
     </zoneMetadata>
    </mapSet>
  </objectgridDeployment>
</deploymentPolicy>
```

Here, you can see two rules, one for the primaries (P) and another for the replica (A).

Zones over wide area networks (WAN)

You might want to deploy a single eXtreme Scale over multiple buildings or data centers with slower network interconnections. Slower network connections lead to lower bandwidth and higher latency connections. The possibility of network partitions also increases in this mode due to network congestion and other factors. eXtreme Scale approaches this harsh environment in the following ways.

Limited heart beating between zones

Java virtual machines grouped together into core groups do heart beat each other. When the catalog service organizes Java virtual machines in to groups, those groups do not span zones. A leader within that group pushes membership information to the catalog service. The catalog service verifies any reported failures before taking action. It does this by attempting to connect to the suspect Java virtual machines. If the catalog service sees a false failure detection then it takes no action as the core group partition will heal in a short period of time.

The catalog service will also heart beat core group leaders periodically at a slow rate to handle the case of core group isolation.

Multi-master data grid replication topologies

Using the multi-master asynchronous replication feature, two or more data grids can become exact mirrors of one other. This mirroring is accomplished using asynchronous replication among links connecting the data grids together. Each data grid is hosted in an independent catalog service domain, with its own catalog service, container servers, and a unique name. With the multi-master asynchronous replication feature, you can use links to interconnect a collection of these catalog service domains. Then, you can synchronize the catalog service domains with replication over the links. You can construct almost any topology because you choose how to define links among catalog service domains.

7.1+ Multi-master data grid replication is a significant new feature in Version 7.1. The feature is also called AP (availability and partitioning) replication in the context of the CAP theorem. The CAP theorem states that a distributed computer system cannot support more than two of the following three properties: consistency, availability, and partition tolerance.

See "Initial considerations for multi-master topologies" on page 23 for map sets that are not replicated.

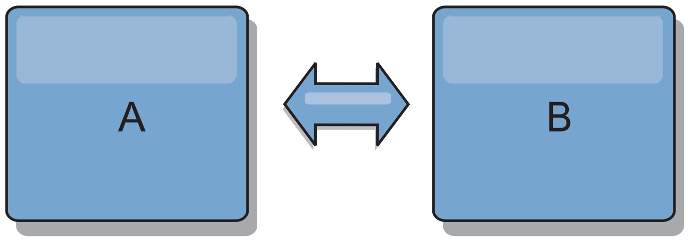
A replication data grid infrastructure is a connected graph of catalog service domains with bidirectional links among them. You can use a link between two catalog service domains to track data changes. For more information about how to set up communication between catalog service domains for multi-master replication, see "Available topologies for multi-master replication" on page 25.

Also, depending on the requirements of your environment, you can optimize the topology design for multi-master replication by taking several factors into consideration: arbitration, linking, and performance. Read more at "Topology considerations for multi-master replication" on page 28.

Available topologies for multi-master replication

You have several different options when choosing the topology for your deployment that incorporates multi-master replication.

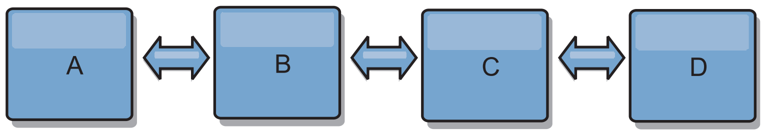
A replication data grid infrastructure is a connected graph of catalog service domains with bidirectional links among them. With a link, two catalog service domains can communicate data changes. For example, the simplest topology is a pair of catalog service domains with a single link between them. The catalog service domains are named alphabetically: A, B, C, and so on, from the left. A link can cross a wide area network (WAN), spanning large distances. Even if the link is interrupted, you can still change data in either catalog service domain. The topology reconciles changes when the link reconnects the catalog service domains. Links automatically try to reconnect if the network connection is interrupted.



After you set up the links, then eXtreme Scale first tries to make every catalog service domain identical. Then, eXtreme Scale tries to maintain the identical conditions as changes occur in any catalog service domain. The goal is for each catalog service domain to be an exact mirror of every other catalog service domain connected by the links. The replication links between the catalog service domains help ensure that any changes made in one domain are copied to the other domains.

Line topologies

Although it is such a simple deployment, a line topology demonstrates some qualities of the links. First, it is not necessary for a catalog service domain to be connected directly to every other catalog service domain to receive changes. Domain B pulls changes from Domain A. Domain C receives changes from Domain A through Domain B, which connects Domains A and C. Similarly, Domain D receives changes from the other domains through Domain C. This ability spreads the load of distributing changes away from the source of the changes.



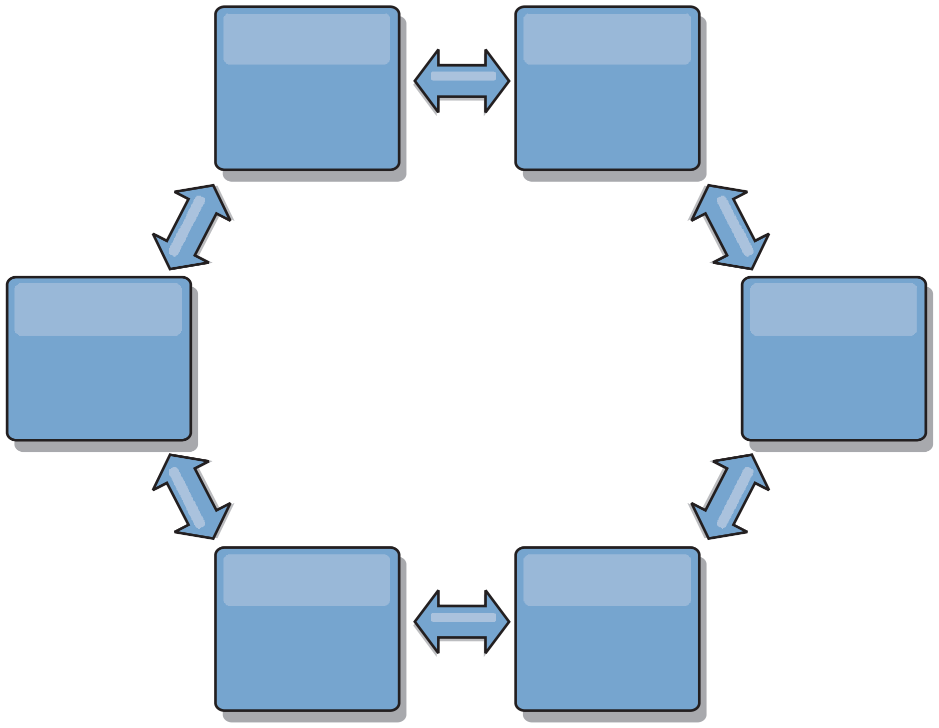
Notice that if Domain C fails, the following would occur:

1. Domain D would be orphaned until Domain C was restarted
2. Domain C would synchronize itself with Domain B, which is a copy of Domain A
3. Domain D would use Domain C to synchronize itself with changes on Domains A and B. These changes initially occurred while Domain D was orphaned (while Domain C was down).

Ultimately, Domains A, B, C, and D would all become identical to one other again.

Ring topologies

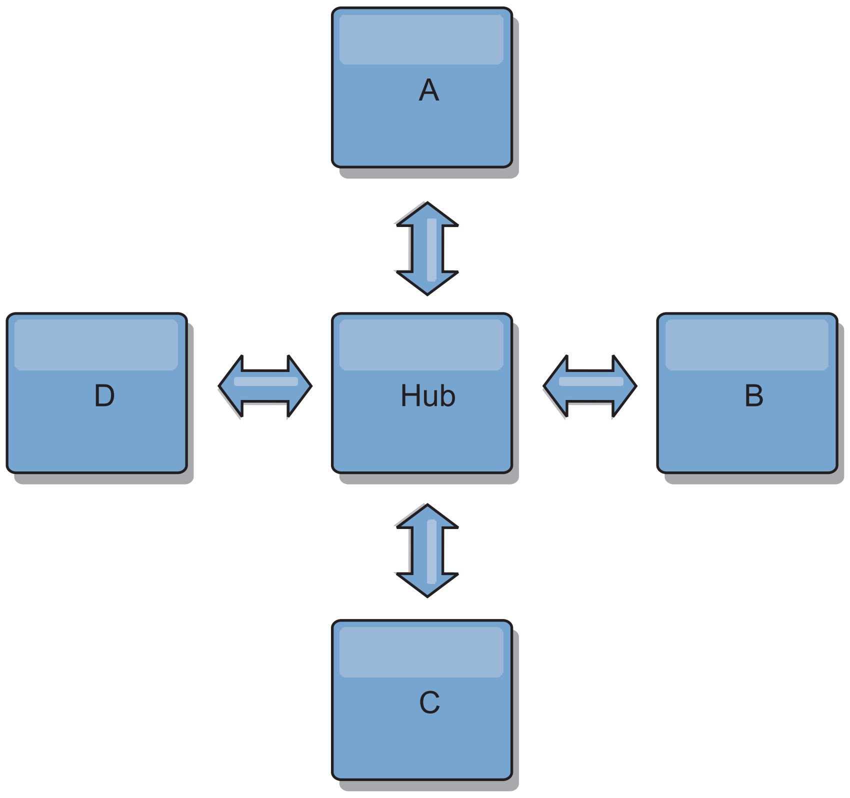
Another option you have with multi-master replication is a ring topology, which is more resilient than the topologies described in the previous sections. A catalog service domain or a single link can fail. Still, the surviving catalog service domains can obtain changes by traveling around the ring, away from the failure. Each catalog service domain has two links to the other catalog service domains. And each catalog service domain has at most two links, no matter how large the ring topology. Changes from a particular domain might travel through several domains before all of them mirror each other. Going through several domains causes potentially high latency, similar to the processes for a line topology.



You can also deploy a more sophisticated ring topology, with a root catalog service domain at the center of the ring. The root catalog service domain functions as the central point of reconciliation. The other catalog service domains act as remote points of reconciliation for changes occurring in the root catalog service domain. The root catalog service domain can arbitrate changes among the catalog service domains. If a ring topology contains more than one ring around a root catalog service domain, the domain can only arbitrate changes among the innermost ring. However, the results of the arbitration spread throughout the catalog service domains in the other rings.

Hub-and-spoke topologies

With a hub-and-spoke topology, changes travel through a hub catalog service domain. Because the hub is the only intermediate catalog service domain that is specified, hub-and-spoke topologies have lower latency. The hub domain is connected to every spoke domain through a link. The hub distributes changes among the catalog service domains. The hub acts as a point of reconciliation for collisions. In an environment with a high update rate, the hub might require run on more hardware than the spokes to remain synchronized. WebSphere eXtreme Scale is designed to scale linearly, meaning you can make the hub larger, as needed, without difficulty. However, if the hub fails, then changes are not distributed until the hub restarts. Any changes on the spoke catalog service domains will be distributed after the hub is reconnected.



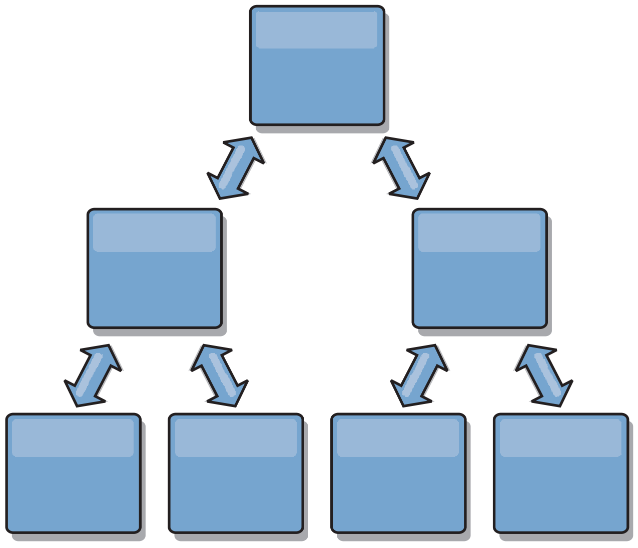
You can also use a strategy with fully replicated clients, a topology variation which uses a pair of eXtreme Scale servers running as a hub. Every client creates a self-contained single container data grid with a catalog in the client JVM. A client uses its data grid to connect to the hub catalog. This connection causes the client to synchronize with the hub as soon as the client obtains a connection to the hub.

Any changes made by the client are local to the client, and are replicated asynchronously to the hub. The hub acts as an arbitration domain, distributing changes to all connected clients. The fully replicated clients topology provides a reliable L2 cache for an object relational mapper, such as OpenJPA. Changes are distributed quickly among client JVMs through the hub. If the cache size can be contained within the available heap space, the topology is a reliable architecture for this style of L2.

Use multiple partitions to scale the hub domain on multiple JVMs, if necessary. Because all of the data still must fit in a single client JVM, multiple partitions increase the capacity of the hub to distribute and arbitrate changes. However, having multiple partitions does not change the capacity of a single domain.

Tree topologies

You can also use an acyclic directed tree. An acyclic tree has no cycles or loops, and a directed setup limits links to existing only between parents and children. You can use the tree topology when you have many catalog service domains such that the ring topology would overwork the hub. You can also use a tree if you require being able to add child catalog service domains without updating the root catalog service domain.



A tree topology can still have a central point of reconciliation in the root catalog service domain. The second level can still function as a remote point of reconciliation for changes occurring in the catalog service domain beneath them. The root catalog service domain can arbitrate changes between the catalog service domains on the second level only. You can also use N-ary trees, each of which have N children at each level. Each catalog service domain connects out to *n* links.

Topology considerations for multi-master replication

When implementing multi-master replication, you must consider aspects in your design such as: arbitration, linking, and performance.

Linking considerations in topology design

Ideally, a topology includes the minimum number of links while optimizing trade-offs among change latency, fault tolerance, and performance characteristics.

- **Change latency**

  Change latency is determined by the number of intermediate catalog service domains a change must go through before arriving at a specific catalog service domain.

  A topology has the best change latency when it eliminates intermediate catalog service domains by linking every catalog service domain to every other catalog service domain. However, a catalog service domain must perform replication work in proportion to its number of links. For large topologies, the sheer number of links to be defined can cause an administrative burden.

  The speed at which a change is copied to other catalog service domains depends on additional factors, such as:

  – Processor and network bandwidth on the source catalog service domain

  – The number of intermediate catalog service domains and links between the source and target catalog service domain

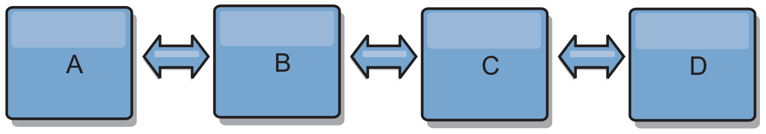
  – The processor and network resources available to the source, target, and intermediate catalog service domains

- **Fault tolerance**

  Fault tolerance is determined by how many paths exist between two catalog service domains for change replication.

If you have only one link between a given pair of catalog service domains, a link failure disallows propagation of changes. Similarly, changes are not propagated between catalog service domains if any of the intermediate domains experiences link failure. Your topology could have a single link from one catalog service domain to another such that the link passes through intermediate domains. If so, then changes are not propagated if any of the intermediate catalog service domains is down.

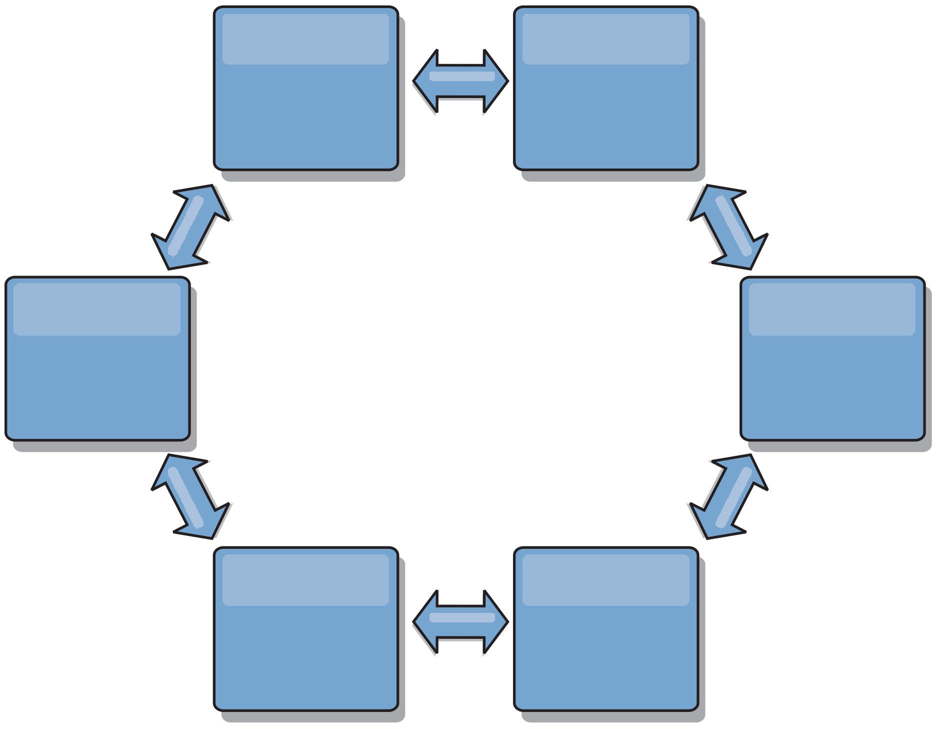
Consider the line topology with four catalog service domains A, B, C, and D:



If any of these conditions hold, Domain D does not see any changes from A:

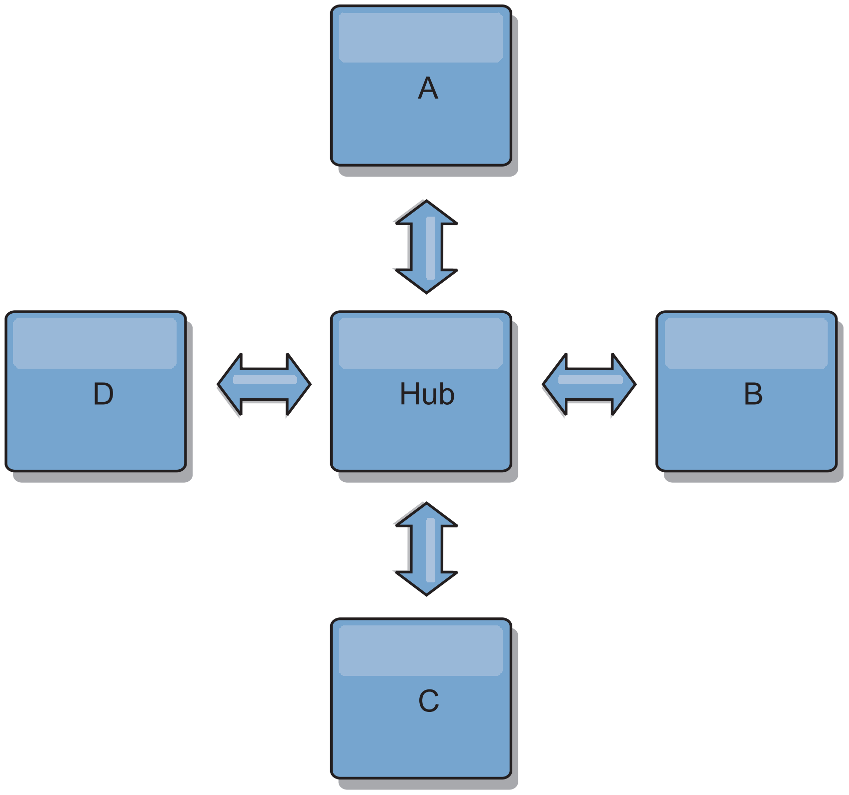
– Domain A is up and B is down
– Domains A and B are up and C is down
– The link between A and B is down
– The link between B and C is down
– The link between C and D is down

In contrast, with a ring topology, each catalog service domain can receive changes from either direction.



For example, if a given catalog service in your ring topology is down, then the two adjacent domains can still pull changes directly from each other.

All changes are propagated through the hub. Thus, as opposed to the line and ring topologies, the hub-and-spoke design is susceptible to breakdown if the hub fails.



A single catalog service domain is resilient to a certain amount of service loss. However, larger failures such as wide network outages or loss of links between physical data centers can disrupt any of your catalog service domains.

- **Linking and performance**

  The number of links defined on a catalog service domain affects performance. More links use more resources and replication performance can drop as a result. The ability to retrieve changes for a domain A through other domains effectively off-loads domain A from replicating its transactions everywhere. The change distribution load on a domain is limited by the number of links it uses, not how many domains are in the topology. This load property provides scalability, so the domains in the topology can share the burden of change distribution.

  A catalog service domain can retrieve changes indirectly through other catalog service domains. Consider a line topology with five catalog service domains.

  ```
  A <=> B <=> C <=> D <=> E
  ```

  – A pulls changes from B, C, D, and E through B
  – B pulls changes from A and C directly, and changes from D and E through C
  – C pulls changes from B and D directly, and changes from A through B and E through D
  – D pulls changes from C and E directly, and changes from A and B through C
  – E pulls changes from D directly, and changes from A, B, and C through D

  The distribution load on catalog service domains A and E is lowest, because they each have a link only to a single catalog service domain. Domains B, C, and D each have a link to two domains. Thus, the distribution load on domains B, C, and D is double the load on domains A and E. The workload depends on the number of links in each domain, not on the overall number of domains in the topology. Thus, the described distribution of loads would remain constant, even if the line contained 1000 domains.

Arbitration considerations in topology design

Change collisions might occur if the same records can be changed simultaneously in two places. Set up each catalog service domain to have about the same amount of processor, memory, network resources. You might observe that catalog service domains performing change collision handling (arbitration) use more resources than other catalog service domains. Collisions are detected automatically. They are handled with one of two mechanisms:

- **Default collision arbiter** The default protocol is to use the changes from the lexically lowest named catalog service domain. For example, if catalog service domain A and B generate a conflict for a record, then the change from catalog service domain B is ignored. Catalog service domain A keeps its version and the record in catalog service domain B is changed to match the record from catalog service domain A. This behavior applies as well for applications where users or sessions are normally bound or have affinity with one of the data grids.
- **Custom collision arbiter** Applications can provide a custom arbiter. When a catalog service domain detects a collision, it starts the arbiter. For information about developing a useful custom arbiter, see Developing custom arbiters for multi-master replication.

For topologies in which collisions are possible, consider implementing a hub-and-spoke topology or a tree topology. These two topologies are conducive to avoiding constant collisions, which can happen in the following scenarios:

1. Multiple catalog service domains experience a collision
2. Each catalog service domain handles the collision locally, producing revisions
3. The revisions collide, resulting in revisions of revisions

To avoid collisions, choose a specific catalog service domain, called an *arbitration catalog service domain* as the collision arbiter for a subset of catalog service domains. For example, a hub-and-spoke topology might use the hub as the collision handler. The spoke collision handler ignores any collisions that are detected by the spoke catalog service domains. The hub catalog service domain creates revisions, preventing unexpected collision revisions. The catalog service domain that is assigned to handle collisions must link to all of the domains for which it is responsible for handling collisions. In a tree topology, any internal parent domains handle collisions for their immediate children. In contrast, if you use a ring topology, you cannot designate one catalog service domain in the ring as the arbiter.

The following table summarizes the arbitration approaches that are most compatible with various topologies.

Table 13. Arbitration approaches. This table states whether application arbitration is compatible with various technologies.

Topology	Application ration?	Notes
A line of two catalog service domains	Yes	Choose one catalog service domain as the arbiter.
A line of three catalog service domains	Yes	The middle catalog service domain must be the arbiter. Think of the middle catalog service domain as the hub in a simple hub-and-spoke topology.
A line of more than three catalog service domains	No	Application arbitration is not supported.

Table 13. Arbitration approaches (continued). This table states whether application arbitration is compatible with various technologies.

Topology	Application ration?	Notes
A hub with N spokes	Yes	Hub with links to all spokes must be the arbitration catalog service domain.
A ring of N catalog service domains	No	Application arbitration is not supported.
An acyclic, directed tree (N-ary tree)	Yes	All root nodes must rate their direct descendants only.

Multi-master replication performance considerations

Take the following limitations into account when using multi-master replication topologies:

- **Change distribution tuning** (Discussed in previous section, "Linking and performance.")
- **Replication link performance** WebSphere eXtreme Scale creates a single TCP/IP socket between any pair of JVMs. All traffic between the JVMs occurs through the single socket, including traffic from multi-master replication. Catalog service domains are hosted on at least *n* container JVMs, providing at least *n* TCP links to peer catalog service domains. Thus, the catalog service domains with larger numbers of containers have higher replication performance levels. More containers require more processor and network resources.
- **TCP sliding window tuning and RFC 1323** RFC 1323 support on both ends of a link yields more data for a round trip. This support results in higher throughput, expanding the capacity of the window by a factor of about 16,000.

  Recall that TCP sockets use a sliding window mechanism to control the flow of bulk data. This mechanism typically limits the socket to 64 KB for a round-trip interval. If the round-trip interval is 100 ms, then the bandwidth is limited to 640 KB/second without additional tuning. Fully using the bandwidth available on a link might require tuning that is specific to an operating system. Most operating systems include tuning parameters, including RFC 1323 options, to enhance throughput over high-latency links.

  Several factors can affect replication performance:
  - The speed at which eXtreme Scale retrieves changes.
  - The speed at which eXtreme Scale can service retrieve replication requests.
  - The sliding window capacity.
  - With network buffer tuning on both sides of a link, eXtreme Scale retrieves changes over the socket efficiently.
- **Object Serialization** All data must be serializable. If a catalog service domain is not using COPY_TO_BYTES, then the catalog service domain must use Java serialization or ObjectTransformers to optimize serialization performance.
- **Compression** WebSphere eXtreme Scale compresses all data sent between catalog service domains by default. Disabling compression is not currently available.
- **Memory tuning** The memory usage for a multi-master replication topology is largely independent of the number of catalog service domains in the topology.

  Multi-master replication adds a fixed overhead per Map entry to handle versioning. Each container also tracks a fixed amount of data for each catalog service domain in the topology. A topology with two catalog service domains uses approximately the same memory as a topology with 50 catalog service domains. WebSphere eXtreme Scale does not use replay logs or similar queues in its implementation. Thus, there is no recovery structure ready in the case that a replication link is unavailable for a substantial period and later restarts.

Distributing transactions

Use Java Message Service (JMS) for distributed transaction changes between different tiers or in environments on mixed platforms.

JMS is an ideal protocol for distributed changes between different tiers or in environments on mixed platforms. For example, some applications that use eXtreme Scale might be deployed on IBM WebSphere Application Server Community Edition, Apache Geronimo, or Apache Tomcat, whereas other applications might run on WebSphere Application Server Version 6.x. JMS is ideal for distributed changes between eXtreme Scale peers in these different environments. The high availability manager message transport is very fast, but can only distribute changes to Java virtual machines that are in a single core group. JMS is slower, but allows larger and more diverse sets of application clients to share an ObjectGrid. JMS is ideal when sharing data in an ObjectGrid between a fat Swing client and an application deployed on WebSphere Extended Deployment.

The built-in Client Invalidation Mechanism and Peer-to-Peer Replication are examples of JMS-based transactional changes distribution. See the information about configuring peer-to-peer replication with JMS in the *Administration Guide* for more information.

Implementing JMS

JMS is implemented for distributing transaction changes by using a Java object that behaves as an ObjectGridEventListener. This object can propagate the state in the following four ways:

1. Invalidate: Any entry that is evicted, updated or deleted is removed on all peer Java virtual machines when they receive the message.
2. Invalidate conditional: The entry is evicted only if the local version is the same or older than the version on the publisher.
3. Push: Any entry that was evicted, updated, deleted or inserted is added or overwritten on all peer Java virtual machines when they receive the JMS message.
4. Push conditional: The entry is only updated or added on the receive side if the local entry is less recent than the version that is being published.

Listen for changes for publishing

The plug-in implements the ObjectGridEventListener interface to intercept the transactionEnd event. When eXtreme Scale invokes this method, the plug-in attempts to convert the LogSequence list for each map that is touched by the transaction to a JMS message and then publish it. The plug-in can be configured to publish changes for all maps or a subset of maps. LogSequence objects are processed for the maps that have publishing enabled. The LogSequenceTransformer ObjectGrid class serializes a filtered LogSequence for each map to a stream. After all LogSequences are serialized to the stream, then a JMS ObjectMessage is created and published to a well-known topic.

Listen for JMS messages and apply them to the local ObjectGrid

The same plug-in also starts a thread that spins in a loop, receiving all messages that are published to the well known topic. When a message arrives, it passes the message contents to the LogSequenceTransformer class where it is converted to a set of LogSequence objects. Then, a no-write-through transaction is started. Each LogSequence object is provided to the Session.processLogSequence method, which updates the local Maps with the changes. The processLogSequence method understands the distribution mode. The transaction is committed and the local cache now reflects the changes. For more information about using JMS to distribute transaction changes, see the information about distributing changes between peer Java Virtual Machines in the *Administration Guide*.

Map sets for replication

Replication is enabled by associating BackingMaps with a MapSet.

A MapSet is a collection of maps that are categorized by partition-key. This partition-key is derived from the individual map's key by taking its hash modulo the number of partitions. Thus, if one group of maps within the MapSet has partition-key X, those maps will be stored in a corresponding partition X in the data grid. If another group has partition-key Y, all of the maps will be stored in partition Y, and so on. Also, the data within the maps is replicated based on the policy defined on the MapSet, which is only used for distributed eXtreme Scale topologies (unnecessary for local instances).

See “Partitioning” on page 83 for more details.

MapSets are assigned what number of partitions they will have and a replication policy. The MapSet replication configuration simply identifies the number of synchronous and asynchronous replica shards a MapSet should have in addition to the primary shard. For example, if there is to be 1 synchronous and 1 asynchronous replica, all of the BackingMaps assigned to the MapSet will each have a replica shard distributed automatically within the set of available containers for the eXtreme Scale. The replication configuration can also enable clients to read data from synchronously replicated servers. This can spread the load for read requests over additional servers in the eXtreme Scale. Replication only has a programming model impact when preloading the BackingMaps.

For details on the various configuration options, see below:

Chapter 6. Transaction processing overview

WebSphere eXtreme Scale uses transactions as its mechanism for interaction with data.

To interact with data, the thread in your application needs its own session. When the application wants to use the ObjectGrid on a thread, call one of the ObjectGrid.getSession methods to obtain a thread. With the session, the application can work with data that is stored in the ObjectGrid maps.

When an application uses a Session object, the session must be in the context of a transaction. A transaction begins and commits or begins and rolls back using the begin, commit, and rollback methods on the Session object. Applications can also work in auto-commit mode, in which the Session automatically begins and commits a transaction whenever an operation is performed on the map. Auto-commit mode cannot group multiple operations into a single transaction, so it is the slower option if you are creating a batch of multiple operations into a single transaction. However, for transactions that contain only one operation, auto-commit is the faster option.

Transactions

Transactions have many advantages for data storage and manipulation. You can use transactions to protect the data grid from concurrent changes, to apply multiple changes as a concurrent unit, to replicate data and to implement a life cycle for locks on changes.

When a transaction starts, WebSphere eXtreme Scale allocates a special difference map to hold the current changes or copies of key and value pairs that the transaction uses. Typically, when a key and value pair is accessed, the value is copied before the application receives the value. The difference map tracks all changes for operations such as insert, update, get, remove, and so on. Keys are not copied because they are assumed to be immutable. If an ObjectTransformer object is specified, then this object is used for copying the value. If the transaction is using optimistic locking, then before images of the values are also tracked for comparison when the transaction commits.

If a transaction is rolled back, then the difference map information is discarded, and locks on entries are released. When a transaction commits, the changes are applied to the maps and locks are released. If optimistic locking is being used, then eXtreme Scale compares the before image versions of the values with the values that are in the map. These values must match for the transaction to commit. This comparison enables a multiple version locking scheme, but at a cost of two copies being made when the transaction accesses the entry. All values are copied again and the new copy is stored in the map. WebSphere eXtreme Scale performs this copy to protect itself against the application changing the application reference to the value after a commit.

You can avoid using several copies of the information. The application can save a copy by using pessimistic locking instead of optimistic locking as the cost of limiting concurrency. The copy of the value at commit time can also be avoided if the application agrees not to change a value after a commit.

Advantages of transactions

Use transactions for the following reasons:

By using transactions, you can:
- Roll back changes if an exception occurs or business logic needs to undo state changes.
- To apply multiple changes as an atomic unit at commit time.
- Hold and release locks on data to apply multiple changes as an atomic unit at commit time.
- Protect a thread from concurrent changes.
- Implement a life cycle for locks on changes.
- Produce an atomic unit of replication.

Transaction size

Larger transactions are more efficient, especially for replication. However, larger transactions can adversely impact concurrency because the locks on entries are held for a longer period of time. If you use larger transactions, you can increase replication performance. This performance increase is important when you are pre-loading a Map. Experiment with different batch sizes to determine what works best for your scenario.

Larger transactions also help with loaders. If a loader is being used that can perform SQL batching, then significant performance gains are possible depending on the transaction and significant load reductions on the database side. This performance gain depends on the Loader implementation.

Automatic commit mode

If no transaction is actively started, then when an application interacts with an ObjectMap object, an automatic begin and commit operation is done on behalf of the application. This automatic begin and commit operation works, but prevents rollback and locking from working effectively. Synchronous replication speed is impacted because of the very small transaction size. If you are using an entity manager application, then do not use automatic commit mode because objects that are looked up with the EntityManager.find method immediately become unmanaged on the method return and become unusable.

External transaction coordinators

Typically, transactions begin with the session.begin method and end with the session.commit method. However, when eXtreme Scale is embedded, the transactions might be started and ended by an external transaction coordinator. If you are using an external transaction coordinator, you do not need to call the session.begin method and end with the session.commit method. If you are using WebSphere Application Server, you can use the WebSphereTranscationCallback plug-in.

CopyMode attribute

You can tune the number of copies by defining the CopyMode attribute of the BackingMap or ObjectMap objects.

You can tune the number of copies by defining the CopyMode attribute of the BackingMap or ObjectMap objects. The copy mode has the following values:

- COPY_ON_READ_AND_COMMIT
- COPY_ON_READ
- NO_COPY
- COPY_ON_WRITE
- COPY_TO_BYTES

The COPY_ON_READ_AND_COMMIT value is the default. The COPY_ON_READ value copies on the initial data retrieved, but does not copy at commit time. This mode is safe if the application does not modify a value after committing a transaction. The NO_COPY value does not copy data, which is only safe for read-only data. If the data never changes then you do not need to copy it for isolation reasons.

Be careful when you use the NO_COPY attribute value with maps that can be updated. WebSphere eXtreme Scale uses the copy on first touch to allow the transaction rollback. The application only changed the copy, and as a result, eXtreme Scale discards the copy. If the NO_COPY attribute value is used, and the application modifies the committed value, completing a rollback is not possible. Modifying the committed value leads to problems with indexes, replication, and so on because the indexes and replicas update when the transaction commits. If you modify committed data and then roll back the transaction, which does not actually roll back at all, then the indexes are not updated and replication does not take place. Other threads can see the uncommitted changes immediately, even if they have locks. Use the NO_COPY attribute value for read-only maps or for applications that complete the appropriate copy before modifying the value. If you use the NO_COPY attribute value and call IBM support with a data integrity problem, you are asked to reproduce the problem with the copy mode set to COPY_ON_READ_AND_COMMIT.

The COPY_TO_BYTES value stores values in the map in a serialized form. At read time, eXtreme Scale inflates the value from a serialized form and at commit time it stores the value to a serialized form. With this method, a copy occurs at both read and commit time.

The default copy mode for a map can be configured on the BackingMap object. You can also change the copy mode on maps before you start a transaction by using the ObjectMap.setCopyMode method.

An example of a backing map snippet from an `objectgrid.xml` file that shows how to set the copy mode for a given backing map follows. This example assumes that you are using `cc` as the `objectgrid/config` namespace.

```
<cc:backingMap name="RuntimeLifespan" copyMode="NO_COPY"/>
```

See the information about copyMode best practices in the *Programming Guide* for more information.

Map entry locking

An ObjectGrid BackingMap supports several locking strategies for maps to maintain cache entry consistency.

Lock manager configuration

When either a PESSIMISTIC or an OPTIMISTIC lock strategy is used, a lock manager is created for the BackingMap. The lock manager uses a hash map to track entries that are locked by one or more transactions. If many map entries exist in the hash map, more lock buckets can result in better performance. The risk of Java synchronization collisions is lower as the number of buckets grows. More lock buckets also lead to more concurrency. The previous examples show how an application can set the number of lock buckets to use for a given BackingMap instance.

To avoid a java.lang.IllegalStateException exception, the setNumberOfLockBuckets method must be called before calling the initialize or getSession methods on the ObjectGrid instance. The setNumberOfLockBuckets method parameter is a Java primitive integer that specifies the number of lock buckets to use. Using a prime number can allow for a uniform distribution of map entries over the lock buckets. A good starting point for best performance is to set the number of lock buckets to about 10 percent of the expected number of BackingMap entries.

Locking strategies

Locking strategies include pessimistic, optimistic and none. To choose a locking strategy, you must consider issues such as the percentage of each type of operations you have, whether or not you use a loader and so on.

Locks are bound by transactions. You can specify the following locking settings:

- **No locking**: Running without the locking setting is the fastest. If you are using read-only data, then you might not need locking.
- **Pessimistic locking**: Acquires locks on entries, then and holds the locks until commit time. This locking strategy provides good consistency at the expense of throughput.
- **Optimistic locking**: Takes a before image of every record that the transaction touches and compares the image to the current entry values when the transaction commits. If the entry values change, then the transaction rolls back. No locks are held until commit time. This locking strategy provides better concurrency than the pessimistic strategy, at the risk of the transaction rolling back and the memory cost of making the extra copy of the entry.

Set the locking strategy on the BackingMap. You cannot change the locking strategy for each transaction. An example XML snippet that shows how to set the locking mode on a map using the XML file follows, assuming `cc` is the namespace for the `objectgrid/config` namespace:

```
<cc:backingMap name="RuntimeLifespan" lockStrategy="PESSIMISTIC" />
```

Pessimistic locking

Use the pessimistic locking strategy for read and write maps when other locking strategies are not possible. When an ObjectGrid map is configured to use the pessimistic locking strategy, a pessimistic transaction lock for a map entry is obtained when a transaction first gets the entry from the BackingMap. The pessimistic lock is held until the application completes the transaction. Typically, the pessimistic locking strategy is used in the following situations:

- When the BackingMap is configured with or without a loader and versioning information is not available.

- When the BackingMap is used directly by an application that needs help from the eXtreme Scale for concurrency control.
- When versioning information is available, but update transactions frequently collide on the backing entries, resulting in optimistic update failures.

Because the pessimistic locking strategy has the greatest impact on performance and scalability, this strategy should only be used for read and write maps when other locking strategies are not viable. For example, these situations might include when optimistic update failures occur frequently, or when recovery from optimistic failure is difficult for an application to handle.

Optimistic locking

The optimistic locking strategy assumes that no two transactions might attempt to update the same map entry while running concurrently. Because of this belief, the lock mode does not need to be held for the life cycle of the transaction because it is unlikely that more than one transaction might update the map entry concurrently. The optimistic locking strategy is typically used in the following situations:

- When a BackingMap is configured with or without a loader and versioning information is available.
- When a BackingMap has mostly transactions that perform read operations. Insert, update, or remove operations on map entries do not occur often on the BackingMap.
- When a BackingMap is inserted, updated, or removed more frequently than it is read, but transactions rarely collide on the same map entry.

Like the pessimistic locking strategy, the methods on the ObjectMap interface determine how eXtreme Scale automatically attempts to acquire a lock mode for the map entry that is being accessed. However, the following differences between the pessimistic and optimistic strategies exist:

- Like the pessimistic locking strategy, an S lock mode is acquired by the get and getAll methods when the method is invoked. However, with optimistic locking, the S lock mode is not held until the transaction is completed. Instead, the S lock mode is released before the method returns to the application. The purpose of acquiring the lock mode is so that eXtreme Scale can ensure that only committed data from other transactions is visible to the current transaction. After eXtreme Scale has verified that the data is committed, the S lock mode is released. At commit time, an optimistic versioning check is performed to ensure that no other transaction has changed the map entry after the current transaction released its S lock mode. If an entry is not fetched from the map before it is updated, invalidated, or deleted, the eXtreme Scale run time implicitly fetches the entry from the map. This implicit get operation is performed to get the current value at the time the entry was requested to be modified.
- Unlike pessimistic locking strategy, the getForUpdate and getAllForUpdate methods are handled exactly like the get and getAll methods when the optimistic locking strategy is used. That is, an S lock mode is acquired at the start of the method and the S lock mode is released before returning to the application.

All other ObjectMap methods are handled exactly like they are handled for the pessimistic locking strategy. That is, when the commit method is invoked, an X lock mode is obtained for any map entry that is inserted, updated, removed, touched, or invalidated and the X lock mode is held until the transaction completes commit processing.

The optimistic locking strategy assumes that no concurrently running transactions attempt to update the same map entry. Because of this assumption, the lock mode does not need to be held for the life of the transaction because it is unlikely that more than one transaction might update the map entry concurrently. However, because a lock mode was not held, another concurrent transaction might potentially update the map entry after the current transaction has released its S lock mode.

To handle this possibility, eXtreme Scale gets an X lock at commit time and performs an optimistic versioning check to verify that no other transaction has changed the map entry after the current transaction read the map entry from the BackingMap. If another transaction changes the map entry, the version check fails and an OptimisticCollisionException exception occurs. This exception forces the current transaction to be rolled back and the application must try the entire transaction again. The optimistic locking strategy is very useful when a map is mostly read and it is unlikely that updates for the same map entry might occur.

No locking

When a BackingMap is configured to use no locking strategy, no transaction locks for a map entry are obtained.

Using no locking strategy is useful when an application is a persistence manager such as an Enterprise JavaBeans (EJB) container or when an application uses Hibernate to obtain persistent data. In this scenario, the BackingMap is configured without a loader and the persistence manager uses the BackingMap as a data cache. In this scenario, the persistence manager provides concurrency control between transactions that are accessing the same Map entries.

WebSphere eXtreme Scale does not need to obtain any transaction locks for the purpose of concurrency control. This situation assumes that the persistence manager does not release its transaction locks before updating the ObjectGrid map with committed changes. If the persistence manager releases its locks, then a pessimistic or optimistic lock strategy must be used. For example, suppose that the persistence manager of an EJB container is updating an ObjectGrid map with data that was committed in the EJB container-managed transaction. If the update of the ObjectGrid map occurs before the persistence manager transaction locks are released, then you can use the no lock strategy. If the ObjectGrid map update occurs after the persistence manager transaction locks are released, then you must use either the optimistic or pessimistic lock strategy.

Another scenario where no locking strategy can be used is when the application uses a BackingMap directly and a Loader is configured for the map. In this scenario, the loader uses the concurrency control support that is provided by a relational database management system (RDBMS) by using either Java database connectivity (JDBC) or Hibernate to access data in a relational database. The loader implementation can use either an optimistic or pessimistic approach. A loader that uses an optimistic locking or versioning approach helps to achieve the greatest amount of concurrency and performance. For more information about implementing an optimistic locking approach, see the OptimisticCallback section in the information about loader considerations in the *Administration Guide*. If you are using a loader that uses pessimistic locking support of an underlying backend, you might want to use the forUpdate parameter that is passed on the get method of the Loader interface. Set this parameter to true if the getForUpdate method of the ObjectMap interface was used by the application to get the data. The loader can use this parameter to determine whether to request an upgradeable lock on the row that is being read. For example, DB2 obtains an upgradeable lock when an SQL select statement contains a `FOR UPDATE` clause. This approach offers the same deadlock prevention that is described in “Pessimistic locking” on page 140.

For more information, see the topic on handling locks in the *Programming Guide* or map entry locking in the *Administration Guide*.

Distributing transactions

Use Java Message Service (JMS) for distributed transaction changes between different tiers or in environments on mixed platforms.

JMS is an ideal protocol for distributed changes between different tiers or in environments on mixed platforms. For example, some applications that use eXtreme Scale might be deployed on IBM WebSphere Application Server Community Edition, Apache Geronimo, or Apache Tomcat, whereas other applications might run on WebSphere Application Server Version 6.x. JMS is ideal for distributed changes between eXtreme Scale peers in these different environments. The high availability manager message transport is very fast, but can only distribute changes to Java virtual machines that are in a single core group. JMS is slower, but allows larger and more diverse sets of application clients to share an ObjectGrid. JMS is ideal when sharing data in an ObjectGrid between a fat Swing client and an application deployed on WebSphere Extended Deployment.

The built-in Client Invalidation Mechanism and Peer-to-Peer Replication are examples of JMS-based transactional changes distribution. See the information about configuring peer-to-peer replication with JMS in the *Administration Guide* for more information.

Implementing JMS

JMS is implemented for distributing transaction changes by using a Java object that behaves as an ObjectGridEventListener. This object can propagate the state in the following four ways:

1. Invalidate: Any entry that is evicted, updated or deleted is removed on all peer Java virtual machines when they receive the message.
2. Invalidate conditional: The entry is evicted only if the local version is the same or older than the version on the publisher.
3. Push: Any entry that was evicted, updated, deleted or inserted is added or overwritten on all peer Java virtual machines when they receive the JMS message.
4. Push conditional: The entry is only updated or added on the receive side if the local entry is less recent than the version that is being published.

Listen for changes for publishing

The plug-in implements the ObjectGridEventListener interface to intercept the transactionEnd event. When eXtreme Scale invokes this method, the plug-in attempts to convert the LogSequence list for each map that is touched by the transaction to a JMS message and then publish it. The plug-in can be configured to publish changes for all maps or a subset of maps. LogSequence objects are processed for the maps that have publishing enabled. The LogSequenceTransformer ObjectGrid class serializes a filtered LogSequence for each map to a stream. After all LogSequences are serialized to the stream, then a JMS ObjectMessage is created and published to a well-known topic.

Listen for JMS messages and apply them to the local ObjectGrid

The same plug-in also starts a thread that spins in a loop, receiving all messages that are published to the well known topic. When a message arrives, it passes the message contents to the LogSequenceTransformer class where it is converted to a set of LogSequence objects. Then, a no-write-through transaction is started. Each LogSequence object is provided to the Session.processLogSequence method, which updates the local Maps with the changes. The processLogSequence method understands the distribution mode. The transaction is committed and the local cache now reflects the changes. For more information about using JMS to distribute transaction changes, see the information about distributing changes between peer Java Virtual Machines in the *Administration Guide*.

Single-partition and cross-data-grid transactions

The major distinction between WebSphere eXtreme Scale and traditional data storage solutions like relational databases or in-memory databases is the use of partitioning, which allows the cache to scale linearly. The important types of transactions to consider are single-partition and every-partition (cross-data-grid) transactions.

In general, interactions with the cache can be categorized as single-partition transactions or cross-data-grid transactions, as discussed in the following section.

Single-partition transactions

Single-partition transactions are the preferable method for interacting with caches that are hosted by WebSphere eXtreme Scale. When a transaction is limited to a single partition, then by default it is limited to a single Java virtual machine, and therefore a single server computer. A server can complete *M* number of these transactions per second, and if you have *N* computers, you can complete `M*N` transactions per second. If your business increases and you need to perform twice as many of these transactions per second, you can double *N* by buying more computers. Then you can meet capacity demands without changing the application, upgrading hardware, or even taking the application offline.

In addition to letting the cache scale so significantly, single-partition transactions also maximize the availability of the cache. Each transaction only depends on one computer. Any of the other `(N-1)` computers can fail without affecting the success or response time of the transaction. So if you are running 100 computers and one of them fails, only 1 percent of the transactions in flight at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale relocates the partitions that are hosted by the failed server to the other 99 computers. During this brief period, before the operation completes, the other 99 computers can still complete transactions. Only the transactions that would involve the partitions that are being relocated are blocked. After the failover process is complete, the cache can continue running, fully operational, at 99 percent of its original throughput capacity. After the failed server is replaced and returned to the data grid, the cache returns to 100 percent throughput capacity.

Cross-data-grid transactions

In terms of performance, availability and scalability, cross-data-grid transactions are the opposite of single-partition transactions. Cross-data-grid transactions access every partition and therefore every computer in the configuration. Each computer in the data grid is asked to look up some data and then return the result. The transaction cannot complete until every computer has responded, and therefore the throughput of the entire data grid is limited by the slowest computer. Adding computers does not make the slowest computer faster and therefore does not improve the throughput of the cache.

Cross-data-grid transactions have a similar effect on availability. Extending the previous example, if you are running 100 servers and one server fails, then 100 percent of the transactions that are in progress at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale starts to relocate the partitions that are hosted by that server to the other 99 computers. During this time, before the failover process completes, the data grid cannot process any of these transactions. After the failover process is complete, the cache can continue running, but at reduced capacity. If each computer in the data grid serviced 10 partitions, then 10 of the remaining 99 computers receive at least one extra partition as part of the failover process. Adding an extra partition increases the workload of that computer by at least 10 percent. Because the throughput of the data grid is limited to the throughput of the slowest computer in a cross-data-grid transaction, on average, the throughput is reduced by 10 percent.

Single-partition transactions are preferable to cross-data-grid transactions for scaling out with a distributed, highly available, object cache like WebSphere eXtreme Scale. Maximizing the performance of these kinds of systems requires the use of techniques that are different from traditional relational methodologies, but you can turn cross-data-grid transactions into scalable single-partition transactions.

Best practices for building scalable data models

The best practices for building scalable applications with products like WebSphere eXtreme Scale include two categories: foundational principles and implementation tips. Foundational principles are core ideas that need to be captured in the design of the data itself. An application that does not observe these principles is unlikely to scale well, even for its mainline transactions. Implementation tips are applied for problematic transactions in an otherwise well-designed application that observes the general principles for scalable data models.

Foundational principles

Some of the important means of optimizing scalability are basic concepts or principles to keep in mind.

Duplicate instead of normalizing

The key thing to remember about products like WebSphere eXtreme Scale is that they are designed to spread data across a large number of computers. If the goal is to make most or all transactions complete on a single partition, then the data model design needs to ensure that all the data the transaction might need is located in the partition. Most of the time, the only way to achieve this is by duplicating data.

For example, consider an application like a message board. Two very important transactions for a message board are showing all the posts from a given user and all the posts on a given topic. First consider how these transactions would work with a normalized data model that contains a user record, a topic record, and a post record that contains the actual text. If posts are partitioned with user records, then displaying the topic becomes a cross-grid transaction, and vice versa. Topics and users cannot be partitioned together because they have a many-to-many relationship.

The best way to make this message board scale is to duplicate the posts, storing one copy with the topic record and one copy with the user record. Then, displaying the posts from a user is a single-partition transaction, displaying the posts on a topic is a single-partition transaction, and updating or deleting a post is a two-partition transaction. All three of these transactions will scale linearly as the number of computers in the data grid increases.

Scalability rather than resources

The biggest obstacle to overcome when considering denormalized data models is the impact that these models have on resources. Keeping two, three, or more copies of some data can seem to use too many resources to be practical. When you are confronted with this scenario, remember the following facts: Hardware resources get cheaper every year. Second, and more importantly, WebSphere eXtreme Scale eliminates most hidden costs associated with deploying more resources.

Measure resources in terms of cost rather than computer terms such as megabytes and processors. Data stores that work with normalized relational data generally need to be located on the same computer. This required collocation means that a single larger enterprise computer needs to be purchased rather than several smaller computers. With enterprise hardware, it is not uncommon for one computer to be capable of completing one million transactions per second to cost much more than the combined cost of 10 computers capable of doing 100,000 transactions per second each.

A business cost in adding resources also exists. A growing business eventually runs out of capacity. When you run out of capacity, you either need to shut down while moving to a bigger, faster computer, or create a second production environment to which you can switch. Either way, additional costs will come in the form of lost business or maintaining almost twice the capacity needed during the transition period.

With WebSphere eXtreme Scale, the application does not need to be shut down to add capacity. If your business projects that you need 10 percent more capacity for the coming year, then increase the number of computers in the data grid by 10 percent. You can increase this percentage without application downtime and without purchasing excess capacity.

Avoid data transformations

When you are using WebSphere eXtreme Scale, data should be stored in a format that is directly consumable by the business logic. Breaking the data down into a more primitive form is costly. The transformation needs to be done when the data is written and when the data is read. With relational databases this transformation is done out of necessity, because the data is ultimately persisted to disk quite frequently, but with WebSphere eXtreme Scale, you do not need to perform these transformations. For the most part data is stored in memory and can therefore be stored in the exact form that the application needs.

Observing this simple rule helps denormalize your data in accordance with the first principle. The most common type of transformation for business data is the JOIN operations that are necessary to turn normalized data into a result set that fits the needs of the application. Storing the data in the correct format implicitly avoids performing these JOIN operations and produces a denormalized data model.

Eliminate unbounded queries

No matter how well you structure your data, unbounded queries do not scale well. For example, do not have a transaction that asks for a list of all items sorted by value. This transaction might work at first when the total number of items is 1000, but when the total number of items reaches 10 million, the transaction returns all 10 million items. If you run this transaction, the two most likely outcomes are the transaction timing out, or the client encountering an out-of-memory error.

The best option is to alter the business logic so that only the top 10 or 20 items can be returned. This logic alteration keeps the size of the transaction manageable no matter how many items are in the cache.

Define schema

The main advantage of normalizing data is that the database system can take care of data consistency behind the scenes. When data is denormalized for scalability, this automatic data consistency management no longer exists. You must implement a data model that can work in the application layer or as a plug-in to the distributed data grid to guarantee data consistency.

Consider the message board example. If a transaction removes a post from a topic, then the duplicate post on the user record needs to be removed. Without a data model, it is possible a developer would write the application code to remove the post from the topic and forget to remove the post from the user record. However, if the developer were using a data model instead of interacting with the cache directly, the removePost method on the data model could pull the user ID from the post, look up the user record, and remove the duplicate post behind the scenes.

Alternately, you can implement a listener that runs on the actual partition that detects the change to the topic and automatically adjusts the user record. A listener might be beneficial because the adjustment to the user record could happen locally if the partition happens to have the user record, or even if the user record is on a different partition, the transaction takes place between servers instead of between the client and server. The network connection between servers is likely to be faster than the network connection between the client and the server.

Avoid contention

Avoid scenarios such as having a global counter. The data grid will not scale if a single record is being used a disproportionate number of times compared to the rest of the records. The performance of the data grid will be limited by the performance of the computer that holds the given record.

In these situations, try to break the record up so it is managed per partition. For example consider a transaction that returns the total number of entries in the distributed cache. Instead of having every insert and remove operation access a single record that increments, have a listener on each partition track the insert and remove operations. With this listener tracking, insert and remove can become single-partition operations.

Reading the counter will become a cross-data-grid operation, but for the most part, it was already as inefficient as a cross-data-grid operation because its performance was tied to the performance of the computer hosting the record.

Implementation tips

You can also consider the following tips to achieve the best scalability.

Use reverse-lookup indexes

Consider a properly denormalized data model where customer records are partitioned based on the customer ID number. This partitioning method is the logical choice because nearly every business operation performed with the customer record uses the customer ID number. However, an important transaction that does not use the customer ID number is the login transaction. It is more common to have user names or e-mail addresses for login instead of customer ID numbers.

The simple approach to the login scenario is to use a cross-data-grid transaction to find the customer record. As explained previously, this approach does not scale.

The next option might be to partition on user name or e-mail. This option is not practical because all the customer ID based operations become cross-data-grid transactions. Also, the customers on your site might want to change their user name or e-mail address. Products like WebSphere eXtreme Scale need the value that is used to partition the data to remain constant.

The correct solution is to use a reverse lookup index. With WebSphere eXtreme Scale, a cache can be created in the same distributed grid as the cache that holds all the user records. This cache is highly available, partitioned and scalable. This cache can be used to map a user name or e-mail address to a customer ID. This cache turns login into a two partition operation instead of a cross-grid operation. This scenario is not as good as a single-partition transaction, but the throughput still scales linearly as the number of computers increases.

Compute at write time

Commonly calculated values like averages or totals can be expensive to produce because these operations usually require reading a large number of entries. Because reads are more common than writes in most applications, it is efficient to compute these values at write time and then store the result in the cache. This practice makes read operations both faster and more scalable.

Optional fields

Consider a user record that holds a business, home, and telephone number. A user could have all, none or any combination of these numbers defined. If the data were normalized then a user table and a telephone number table would exist. The telephone numbers for a given user could be found using a JOIN operation between the two tables.

De-normalizing this record does not require data duplication, because most users do not share telephone numbers. Instead, empty slots in the user record must be allowed. Instead of having a telephone number table, add three attributes to each user record, one for each telephone number type. This addition of attributes eliminates the JOIN operation and makes a telephone number lookup for a user a single-partition operation.

Placement of many-to-many relationships

Consider an application that tracks products and the stores in which the products are sold. A single product is sold in many stores, and a single store sells many products. Assume that this application tracks 50 large retailers. Each product is sold in a maximum of 50 stores, with each store selling thousands of products.

Keep a list of stores inside the product entity (arrangement A), instead of keeping a list of products inside each store entity (arrangement B). Looking at some of the transactions this application would have to perform illustrates why arrangement A is more scalable.

First look at updates. With arrangement A, removing a product from the inventory of a store locks the product entity. If the data grid holds 10000 products, only 1/10000 of the grid needs to be locked to perform the update. With arrangement B, the data grid only contains 50 stores, so 1/50 of the grid must be locked to complete the update. So even though both of these could be considered single-partition operations, arrangement A scales out more efficiently.

Now, considering reads with arrangement A, looking up the stores at which a product is sold is a single-partition transaction that scales and is fast because the transaction only transmits a small amount of data. With arrangement B, this transaction becomes an cross-data-grid transaction because each store entity must be accessed to see if the product is sold at that store, which reveals an enormous performance advantage for arrangement A.

Scaling with normalized data

One legitimate use of cross-data-grid transactions is to scale data processing. If a data grid has 5 computers and a cross-data-grid transaction is dispatched that sorts through about 100,000 records on each computer, then that transaction sorts through 500,000 records. If the slowest computer in the data grid can perform 10 of these transactions per second, then the data grid is capable of sorting through 5,000,000 records per second. If the data in the grid doubles, then each computer must sort through 200,000 records, and each transaction sorts through 1,000,000 records. This data increase decreases the throughput of the slowest computer to 5 transactions per second, thereby reducing the throughput of the data grid to 5 transactions per second. Still, the data grid sorts through 5,000,000 records per second.

In this scenario, doubling the number of computer allows each computer to return to its previous load of sorting through 100,000 records, allowing the slowest computer to process 10 of these transactions per second. The throughput of the data grid stays the same at 10 requests per second, but now each transaction processes 1,000,000 records, so the grid has doubled its capacity in terms of processing records to 10,000,000 per second.

Applications such as a search engine that need to scale both in terms of data processing to accommodate the increasing size of the Internet and throughput to accommodate growth in the number of users, you must create multiple data grids, with a round robin of the requests between the grids. If you need to scale up the throughput, add computers and add another data grid to service requests. If data processing needs to be scaled up, add more computers and keep the number of data grids constant.

Chapter 7. Security overview

WebSphere eXtreme Scale can secure data access, including allowing for integration with external security providers.

Note: In an existing non-cached data store such as a database, you likely have built-in security features that you might not need to actively configure or enable. However, after you have cached your data with eXtreme Scale, you must consider the important resulting situation that your backend security features are no longer in effect. You can configureeXtreme Scale security on necessary levels so that your new cached architecture for your data is also secured.
A brief summary of eXtreme Scale security features follows. For more detailed information about configuring security see the *Administration Guide* and the *Programming Guide*.

Distributed security basics

Distributed eXtreme Scale security is based on three key concepts:

Trustable authentication
: The ability to determine the identity of the requester. WebSphere eXtreme Scale supports both client-to-server and server-to-server authentication.

Authorization
: The ability to give permissions to grant access rights to the requester. WebSphere eXtreme Scale supports different authorizations for various operations.

Secure transport
: The safe transmission of data over a network. WebSphere eXtreme Scale supports the Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocols.

Authentication

WebSphere eXtreme Scale supports a distributed client server framework. A client server security infrastructure is in place to secure access to eXtreme Scale servers. For example, when authentication is required by the eXtreme Scale server, an eXtreme Scale client must provide credentials to authenticate to the server. These credentials can be a user name and password pair, a client certificate, a Kerberos ticket, or data that is presented in a format that is agreed upon by client and server.

Authorization

WebSphere eXtreme Scale authorizations are based on subjects and permissions. You can use the Java Authentication and Authorization Services (JAAS) to authorize the access, or you can plug in a custom approach, such as Tivoli® Access Manager (TAM), to handle the authorizations. The following authorizations can be given to a client or group:

Map authorization
: Perform insert, read, update, evict, or delete operations on Maps.

ObjectGrid authorization
Perform object or entity queries and stream queries on ObjectGrid objects.

DataGrid agent authorization
Allow DataGrid agents to be deployed to an ObjectGrid.

Server side map authorization
Replicate a server map to client side or create a dynamic index to the server map.

Administration authorization
Perform administration tasks.

Transport security

To secure the client server communication, WebSphere eXtreme Scale supports TLS/SSL. These protocols provide transport layer security with authenticity, integrity, and confidentiality for a secure connection between an eXtreme Scale client and server.

Grid security

In a secure environment, a server must be able to check the authenticity of another server. WebSphere eXtreme Scale uses a shared secret key string mechanism for this purpose. This secret key mechanism is similar to a shared password. All the eXtreme Scale servers agree on a shared secret string. When a server joins the data grid, the server is challenged to present the secret string. If the secret string of the joining server matches the one in the master server, then the joining server can join the grid. Otherwise, the join request is rejected.

Sending a clear text secret is not secure. The eXtreme Scale security infrastructure provides a SecureTokenManager plug-in to allow the server to secure this secret before sending it. You can choose how you implement the secure operation. WebSphere eXtreme Scale provides an implementation, in which the secure operation is implemented to encrypt and sign the secret.

Java Management Extensions (JMX) security in a dynamic deployment topology

JMX MBean security is supported in all versions of eXtreme Scale. Clients of catalog server MBeans and container server MBeans can be authenticated, and access to MBean operations can be enforced.

Local eXtreme Scale security

Local eXtreme Scale security is different from the distributed eXtreme Scale model because the application directly instantiates and uses an ObjectGrid instance. Your application and eXtreme Scale instances are in the same Java virtual machine (JVM). Because no client-server concept exists in this model, authentication is not supported. Your applications must manage their own authentication, and then pass the authenticated Subject object to the eXtreme Scale. However, the authorization mechanism that is used for the local eXtreme Scale programming model is the same as what is used for the client-server model.

Configuration and programming

For more information about configuring and programming for security, see the *Administration Guide* and *Programming Guide*.

Chapter 8. REST data services overview

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). Microsoft WCF Data Services is compatible with this specification when using Visual Studio 2008 SP1 and the .NET Framework 3.5 SP1.

Compatibility requirements

The REST data service allows any HTTP client to access a data grid. The REST data service is compatible with the WCF Data Services support 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. The figure provides an overview of how WCF Data Services interacts with clients and databases.

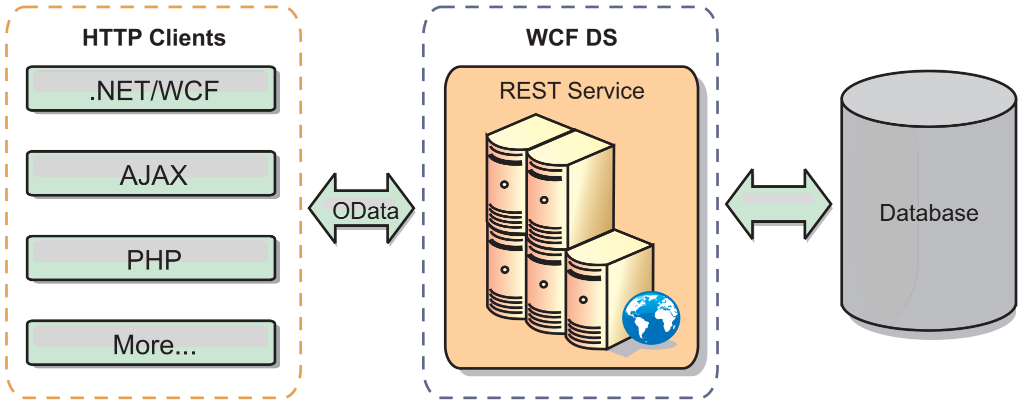


Figure 40. Microsoft WCF Data Services

WebSphere eXtreme Scale includes a function-rich API set for Java clients. As shown in the following figure, the REST data service is a gateway between HTTP clients and the WebSphere eXtreme Scale data grid, communicating with the grid through an WebSphere eXtreme Scale client. The REST data service is a Java servlet, which allows flexible deployments for common Java Platform, Enterprise Edition (JEE) platforms, such as WebSphere Application Server. The REST data service communicates with the WebSphere eXtreme Scale data grid using the WebSphere eXtreme Scale Java APIs. It allows WCF Data Services clients or any other client that can communicate with HTTP and XML.

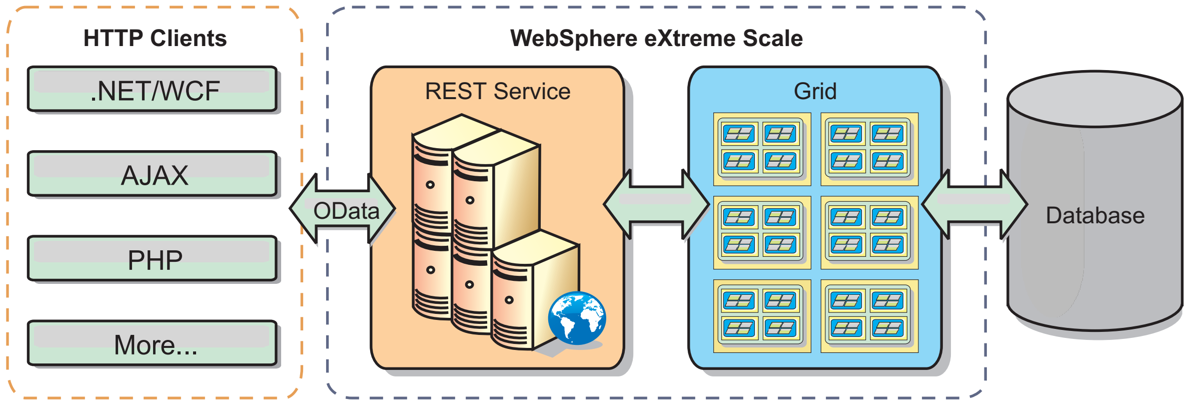


Figure 41. WebSphere eXtreme Scale REST data service

Refer to the "REST data services sample and tutorial" on page 199, or use the following links to learn more about WCF Data Services.

- Microsoft WCF Data Services Developer Center
- ADO.NET Data Services overview on MSDN
- Whitepaper: Using ADO.NET Data Services
- Atom Publish Protocol: Data Services URI and Payload Extensions
- Conceptual Schema Definition File Format
- Entity Data Model for Data Services Packaging Format
- Open Data Protocol
- Open Data Protocol FAQ

Features

This version of the eXtreme Scale REST data service supports the following features:

- Automatic modeling of eXtreme Scale EntityManager API entities as WCF Data Services entities, which includes the following support:
  - Java data type to Entity Data Model type conversion
  - Entity association support
  - Schema root and key association support, which is required for partitioned data grids

  See Entity model for more information.
- Atom Publish Protocol (AtomPub or APP) XML and JavaScript Object Notation (JSON) data payload format.
- Create, Read, Update and Delete (CRUD) operations using the respective HTTP request methods: POST, GET, PUT and DELETE. In addition, the Microsoft extension: MERGE is supported.
- Simple queries, using filters
- Batch retrieval and change set requests
- Partitioned data grid support for high availability
- Interoperability with eXtreme Scale EntityManager API clients
- Support for standard JEE Web servers
- Optimistic concurrency
- User authorization and authentication between the REST data service and the eXtreme Scale data grid

Known problems and limitations

- Tunneled requests are not supported.

Chapter 9. Spring framework

Spring is a framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage 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 use 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.

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 results 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 an HTTP session by default. If a web application uses eXtreme Scale for session management, then Spring automatically stores state with eXtreme Scale. Also, fault tolerance is enabled in the same manner as the session.

See the ObjectMap API information in the *Product Overview* for further details.

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 Java EE application that is running in a WebSphere Extended Deployment augmented

WebSphere Application Server Network Deployment, put 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.

Chapter 10. Tutorials, examples, and samples

Several WebSphere eXtreme Scale tutorials, examples, and samples are available.

Tutorials

The following tutorials are currently available.

- "Entity manager tutorial: Overview" on page 167
-
-

Examples

The following topics illustrate key WebSphere eXtreme Scale features.

- See the Data Grid API example in the *Programming Guide*
- See the details on configuring local deployments in the *Administration Guide*

Community samples

The following samples illustrate how to use WebSphere eXtreme Scale in various environments to exhibit different features of the product.

- **xsadmin utility** - With the xsadmin sample utility, you can format and display textual information about your WebSphere® eXtreme Scale topology. The sample utility provides a method for parsing and discovering current deployment data, and can be used as a foundation for writing custom utilities.

  For more information, see the information about monitoring with the xsAdmin sample utility in the *Administration Guide*. (Shipped with product.)
- **Asynchronous Service Framework** - The Asynchronous Service framework provides a scalable and fault-tolerant processing fabric for asynchronous processing of messages. For more information, including how to download the sample, see the Samples Gallery: Asynchronous Service Framework sample .
- **Client authentication security** - This sample describes how to configure authentication requiring the client to provide valid credentials before the server gives any grid access. For more information, including how to download the sample, see the Samples Gallery: Client authentication security
- **Creating dynamic maps** - This sample demonstrates how to create maps after your grid has already been initialized. For eXtreme Scale 7.0 and higher, you can use templates to retrieve maps. For more information, including how to download the sample, see the Samples Gallery: Creating dynamic maps after grid initialization .
- **Multi-master replication** - The Multi-Master Replication Getting Started sample is provided for a quick introduction to multi-master (AP) replication. For more information, including how to download the sample, see the Samples Gallery: Multi-master Replication sample.
- **Queries with Entity Manager API** - This sample demonstrates how to use queries in a distributed partitioned map with the EntityManager API. For more information, including how to download the sample, see the Samples Gallery: Running Queries in a partitioned grid using Entity Manager API .
- **Parallel queries with a ReduceGridAgent implementation** - Demonstrates how to use the Data Grid API to run a query over every partition in the grid. For

more information, including how to download the sample, see the Samples Gallery: Running Queries in Parallel using a ReduceGridAgent .

Articles with tutorials and examples

Table 14. Available articles by feature

Article	Features
Building grid-ready applications	ObjectMap API, EntityManager API, Query, Agents, Java SE and EE, Statistics, Partitioning, Administration/Operations, Eclipse
Scalable grid-style computing and data processing	EntityManager API, Agents
Building a scalable, resilient, high-performance database alternative	ObjectMap API, Replication, Partitioning, Administration/Operations, Eclipse
Enhancing xsadmin for WebSphere eXtreme Scale	Administration
Redbook: User's Guide	All topics

Running the getting started sample application

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 data grid and client can be used. The getting started sample is in the *wxs_install_root*`/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 data 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 data 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 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` *`wxs_install_root`*`/ObjectGrid/gettingstarted`
   Substitute *wxs_install_root* with the path to the eXtreme Scale installation root directory or the root file path of the extracted eXtreme Scale trial *wxs_install_root*.
3. 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.
4. 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 this step with a different server name if you want to start more container server instances to support replication.
5. 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`

6. 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 *wxs_install_root*`/ObjectGrid/gettingstarted/xml` directory to start a container server. The `objectgrid.xml` file is the ObjectGrid descriptor XML file. The `deployment.xml` file is the ObjectGrid deployment policy descriptor XML file. These 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 startup. 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 data 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 *wxs_install_root*`/ObjectGrid/gettingstarted/client/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. **Connect 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 requires only the catalog server endpoint in the format of *hostname*:*port*. You can indicate multiple catalog server endpoints by separating the list of *hostname*:*port* values with commas. 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);
   ```

   If the connections to the catalog servers succeed, the connect method returns a ClientClusterContext instance. The ClientClusterContext instance is required to obtain the ObjectGrid from ObjectGridManager API.
2. **Obtain 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 data grid instance. The ClientClusterContext instance is obtained during the connection to catalog server. The name of ObjectGrid instance is `Grid` that is specified in the `objectgrid.xml` file. The following code snippet demonstrates how to obtain the data grid by calling the getObjectGrid method of the ObjectGridManager API.

   ```
   ObjectGrid grid = ObjectGridManagerFactory.getObjectGridManager().getObjectGrid(ccc, "Grid");
   ```
3. **Get a Session instance.**

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

   ```
   Session sess = grid.getSession();
   ```
4. **Get an ObjectMap instance.**

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

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

   ```
   ObjectMap map1 = sess.getMap("my_simple_data_grid");
   ```
5. **Use the ObjectMap methods.**

   After an ObjectMap instance is obtained, you can use the ObjectMap API. Remember that the ObjectMap interface 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 in the application, the ObjectMap operations run with auto-commit transactions.

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

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

   The following code snippet demonstrates how to use the 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 deployment manager process and each WebSphere Application Server instance hosts a container 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.

Entity manager tutorial: Overview

The tutorial for the entity manager shows you how to use WebSphere eXtreme Scale to store order information on a Web site. You can create a simple Java Platform, Standard Edition 5 application that uses an in-memory, local eXtreme Scale. The entities use Java SE 5 annotations and generics.

Before you begin

Ensure that you have met the following requirements before you begin the tutorial:
- You must have Java SE 5.
- You must have the `objectgrid.jar` file in your classpath.

Entity manager tutorial: Creating an entity class

The first step of the entity manager tutorial shows you how to create a local ObjectGrid with one entity by creating an Entity class, registering the entity type with eXtreme Scale, and storing an entity instance into the cache.

About this task

Procedure

1. Create the Order object. To identify the object as an ObjectGrid entity, add the @Entity annotation. When you add this annotation, all serializable attributes in the object are automatically persisted in eXtreme Scale, unless you use annotations on the attributes to override the attributes. The orderNumber attribute is annotated with @Id to indicate that this attribute is the primary key. An example of an Order object follows:

   `Order.java`

   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       Date date;
       String customerName;
       String itemName;
       int quantity;
       double price;
   }
   ```

2. Run the eXtreme Scale Hello World application to demonstrate the entity operations. The following example program can be issued in stand-alone mode to demonstrate the entity operations. Use this program in an Eclipse Java project that has the `objectgrid.jar` file added to the class path. An example of a simple Hello world application that uses eXtreme Scale follows:

   Application.java

```
package emtutorial.basic.step1;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Order.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Order o = new Order();
        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: " + o.customerName);
        em.getTransaction().commit();
    }
}
```

   This example application performs the following operations:

   a. Initializes a local eXtreme Scale with an automatically generated name.
   b. Registers the entity classes with the application by using the registerEntities API, although using the registerEntities API is not always necessary.
   c. Retrieves a Session and a reference to the entity manager for the Session.
   d. Associates each eXtreme Scale Session with a single EntityManager and EntityTransaction. The EntityManager is now used.
   e. The registerEntities method creates a BackingMap object that is called Order, and associates the metadata for the Order object with the BackingMap object. This metadata includes the key and non-key attributes, along with the attribute types and names.
   f. A transaction starts and creates an Order instance. The transaction is populated with some values, and is persisted by using the EntityManager.persist method, which identifies the entity as waiting to be included in the associated ObjectGrid Map.
   g. The transaction is then committed, and the entity is included in the ObjectMap.
   h. Another transaction is made, and the Order object is retrieved by using the key 1. The type cast on the EntityManager.find method is necessary, because Java SE 5 generics capability are not used to ensure that the `objectgrid.jar` file works on a Java SE 1.4 and later Java virtual machine.

Entity manager tutorial: Forming entity relationships

Create a simple relationship between entities by creating two entity classes with a relationship, registering the entities with the ObjectGrid, and storing the entity instances into the cache.

Procedure

1. Create the customer entity, which is used to store customer details independently from the Order object. An example of the customer entity follows:

   Customer.java

   ```
   @Entity
   public class Customer
   {
       @Id String id;
       String firstName;
       String surname;
       String address;
       String phoneNumber;
   }
   ```

   This class includes information about the customer such as name, address, and phone number.

2. Create the Order object, which is similar to the Order object in the “Entity manager tutorial: Creating an entity class” on page 167 topic. An example of the order object follows:

   Order.java

   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       Date date;
       @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
       String itemName;
       int quantity;
       double price;
   }
   ```

   In this example, a reference to a Customer object replaces the customerName attribute. The reference has an annotation that indicates a many-to-one relationship. A many-to-one relationship indicates that each order has one customer, but multiple orders might reference the same customer. The cascade annotation modifier indicates that if the entity manager persists the Order object, it must also persist the Customer object. If you choose to not set the cascade persist option, which is the default option, you must manually persist the Customer object with the Order object.

3. Using the entities, define the maps for the ObjectGrid instance. Each map is defined for a specific entity, and one entity is named Order and the other is named Customer. The following example application illustrates how to store and retrieve a customer order:

   Application.java

   ```
   public class Application
   {
       static public void main(String [] args)
           throws Exception
       {
           ObjectGrid og =
        ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
           og.registerEntities(new Class[] {Order.class});
   ```

```
        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";

        Order o = new Order();
        o.customer = cust;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: "
      + o.customer.firstName + " " + o.customer.surname);
        em.getTransaction().commit();
    }
}
```

This application is similar to the example application that is in the previous step. In the preceding example, only a single class Order is registered. WebSphere eXtreme Scale detects and automatically includes the reference to the Customer entity, and a Customer instance for `John Smith` is created and referenced from the new Order object. As a result, the new customer is automatically persisted, because the relationship between two orders includes the cascade modifier, which requires that each object be persisted. When the Order object is found, the entity manager automatically finds the associated Customer object and inserts a reference to the object.

Entity manager tutorial: Order Entity Schema

Create four entity classes by using both single and bidirectional relationships, ordered lists, and foreign key relationships. The EntityManager APIs are used to persist and find the entities. Building on the Order and Customer entities that are in the previous parts of the tutorial, this tutorial step adds two more entities: the Item and OrderLine entities.

About this task

Figure 42. Order Entity Schema. An Order entity has a reference to one customer and zero or more OrderLines. Each OrderLine entity has a reference to a single item and includes the quantity ordered.

Procedure

1. Create the customer entity, which is similar to the previous examples.

   `Customer.java`
   ```
   @Entity
   public class Customer
   {
       @Id String id;
       String firstName;
       String surname;
       String address;
       String phoneNumber;
   }
   ```

2. Create the Item entity, which holds information about a product that is included in the store's inventory, such as the product description, quantity, and price.

   Item.java
   ```
   @Entity
   public class Item
   {
       @Id String id;
       String description;
       long quantityOnHand;
       double price;
   }
   ```
3. Create the OrderLine entity. Each Order has zero or more OrderLines, which identify the quantity of each item in the order. The key for the OrderLine is a compound key that consists of the Order that owns the OrderLine and an integer that assigns the order line a number. Add the cascade persist modifier to every relationship on your entities.

   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;
   }
   ```
4. Create the final Order Object, which has a reference to the Customer for the order and a collection of OrderLine objects.

   Order.java
   ```
   @Entity
   public class Order
   {
       @Id String orderNumber;
       java.util.Date date;
       @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
       @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
      @OrderBy("lineNumber") List<OrderLine> lines;
   }
   ```
   The cascade ALL is used as the modifier for lines. This modifier signals the EntityManager to cascade both the PERSIST operation and the REMOVE operation. For example, if the Order entity is persisted or removed, then all OrderLine entities are also persisted or removed.

   If an OrderLine entity is removed from the lines list in the Order object, the reference is then broken. However, the OrderLine entity is not removed from the cache. You must use the EntityManager remove API to remove entities from the cache. The REMOVE operation is not used on the customer entity or the item entity from OrderLine. As a result, the customer entity remains even though the order or item is removed when the OrderLine is removed.

   The mappedBy modifier indicates an inverse relationship with the target entity. The modifier identifies which attribute in the target entity references the source entity, and the owning side of a one-to-one or many-to-many relationship. Typically, you can omit the modifier. However, an error is displayed to indicate that it must be specified if WebSphere eXtreme Scale cannot discover it automatically. An OrderLine entity that contains two of type Order attributes in a many-to-one relationship typically causes the error.

   The @OrderBy annotation specifies the order in which each OrderLine entity should be in the lines list. If the annotation is not specified, then the lines display in an arbitrary order. Although the lines are added to the Order entity by issuing ArrayList, which preserves the order, the EntityManager does not necessarily recognize the order. When you issue the find method to retrieve the Order object from the cache, the list object is not an ArrayList object.

5. Create the application. The following example illustrates the final Order object, which has a reference to the Customer for the order and a collection of OrderLine objects.
   a. Find the Items to order, which then become Managed entities.
   b. Create the OrderLine and attach it to each Item.
   c. Create the Order and associate it with each OrderLine and the customer.
   d. Persist the order, which automatically persists each OrderLine.
   e. Commit the transaction, which detaches each entity and synchronizes the state of the entities with the cache.
   f. Print the order information. The OrderLine entities are automatically sorted by the OrderLine ID.

```
Application.java

static public void main(String [] args)
        throws Exception
    {
        ...

        // Add some items to our inventory.
        em.getTransaction().begin();
        createItems(em);
        em.getTransaction().commit();

        // Create a new customer with the items in his cart.
        em.getTransaction().begin();
        Customer cust = createCustomer();
        em.persist(cust);

        // Create a new order and add an order line for each item.
        // Each line item is automatically persisted since the
    // Cascade=ALL option is set.
        Order order = createOrderFromItems(em, cust, "ORDER_1",
     new String[]{"1", "2"}, new int[]{1,3});
        em.persist(order);
        em.getTransaction().commit();

        // Print the order summary
        em.getTransaction().begin();
        order = (Order)em.find(Order.class, "ORDER_1");
        System.out.println(printOrderSummary(order));
        em.getTransaction().commit();
    }

    public static Customer createCustomer() {
        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        return cust;
    }

    public static void createItems(EntityManager em) {
        Item item1 = new Item();
        item1.id = "1";
        item1.price = 9.99;
        item1.description = "Widget 1";
        item1.quantityOnHand = 4000;
        em.persist(item1);

        Item item2 = new Item();
        item2.id = "2";
        item2.price = 15.99;
        item2.description = "Widget 2";
        item2.quantityOnHand = 225;
        em.persist(item2);

    }

    public static Order createOrderFromItems(EntityManager em,
   Customer cust, String orderId, String[] itemIds, int[] qty) {

        Item[] items = getItems(em, itemIds);

        Order order = new Order();
        order.customer = cust;
        order.date = new java.util.Date();
        order.orderNumber = orderId;
        order.lines = new ArrayList<OrderLine>(items.length);
    for(int i=0;i<items.length;i++){
      OrderLine line = new OrderLine();
            line.lineNumber = i+1;
            line.item = items[i];
            line.price = line.item.price;
            line.quantity = qty[i];
            line.order = order;
            order.lines.add(line);
        }
        return order;
    }

    public static Item[] getItems(EntityManager em, String[] itemIds) {
        Item[] items = new Item[itemIds.length];
        for(int i=0;i<items.length;i++){
    items[i] = (Item) em.find(Item.class, itemIds[i]);
        }
        return items;
    }
```

The next step is to delete an entity. The EntityManager interface has a remove method that marks an object as deleted. The application should remove the entity from any relationship collections before calling the remove method. Edit the references and issue the remove method, or em.remove(object), as a final step.

Entity manager tutorial: Updating entries

If you want to change an entity, you can find the instance, update the instance and any referenced entities, and commit the transaction.

Procedure

Update entries. The following example demonstrates how to find the Order instance, change it and any referenced entities, and commit the transaction.

```
public static void updateCustomerOrder(EntityManager em) {
    em.getTransaction().begin();
    Order order = (Order) em.find(Order.class, "ORDER_1");
    processDiscount(order, 10);
    Customer cust = order.customer;
    cust.phoneNumber = "5075551234";
    em.getTransaction().commit();
}
```

```
public static void processDiscount(Order order, double discountPct) {
    for(OrderLine line : order.lines) {
        line.price = line.price * ((100-discountPct)/100);
    }
}
```

Flushing the transaction synchronizes all managed entities with the cache. When a transaction is committed, a flush automatically occurs. In this case, the Order becomes a managed entity. Any entities that are referenced from the Order, Customer, and OrderLine also become managed entities. When the transaction is flushed, each of the entities are checked to determine if they have been modified. Those that are modified are updated in the cache. After the transaction completes, by either being committed or rolled back, the entities become detached and any changes that are made in the entities are not reflected in the cache.

Entity manager tutorial: Updating and removing entries with an index

You can use an index to find, update, and remove entities.

Procedure

Update and remove entities by using an index. Use an index to find, update, and remove entities. In the following examples, the Order entity class is updated to use the @Index annotation. The @Index annotation signalsWebSphere eXtreme Scale to create a range index for an attribute. The name of the index is the same name as the name of the attribute and is always a MapRangeIndex index type.

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
    @OrderBy("lineNumber") List<OrderLine> lines;  }
```

The following example demonstrates how to cancel all orders that are submitted within the last minute. Find the order by using an index, add the items in the order back into the inventory, and remove the order and the associated line items from the system.

```
public static void cancelOrdersUsingIndex(Session s)
  throws ObjectGridException {
    // Cancel all orders that were submitted 1 minute ago
    java.util.Date cancelTime = new
   java.util.Date(System.currentTimeMillis() - 60000);
    EntityManager em = s.getEntityManager();
    em.getTransaction().begin();
    MapRangeIndex dateIndex = (MapRangeIndex)
   s.getMap("Order").getIndex("date");
    Iterator<Tuple> orderKeys = dateIndex.findGreaterEqual(cancelTime);
  while(orderKeys.hasNext()) {
   Tuple orderKey = orderKeys.next();
   // Find the Order so we can remove it.
   Order curOrder = (Order) em.find(Order.class, orderKey);
   // Verify that the order was not updated by someone else.
   if(curOrder != null && curOrder.date.getTime() >= cancelTime.getTime()) {
    for(OrderLine line : curOrder.lines) {
     // Add the item back to the inventory.
     line.item.quantityOnHand += line.quantity;
     line.quantity = 0;
    }
   em.remove(curOrder);
```

```
    }
  }
 em.getTransaction().commit();
}
```

Entity manager tutorial: Updating and removing entries by using a query

You can update and remove entities by using a query.

Procedure

Update and remove entities by using a query.

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
   @OrderBy("lineNumber") List<OrderLine> lines;
}
```

The order entity class is the same as it is in the previous example. The class still provides the @Index annotation, because the query string uses the date to find the entity. The query engine uses indices when they can be used.

```
public static void cancelOrdersUsingQuery(Session s) {
        // Cancel all orders that were submitted 1 minute ago
        java.util.Date cancelTime =
     new java.util.Date(System.currentTimeMillis() - 60000);
        EntityManager em = s.getEntityManager();
        em.getTransaction().begin();

        // Create a query that will find the order based on date.  Since
        // we have an index defined on the order date, the query
    // will automatically use it.
        Query query = em.createQuery("SELECT order FROM Order order
     WHERE order.date >= ?1");
        query.setParameter(1, cancelTime);
        Iterator<Order> orderIterator = query.getResultIterator();
    while(orderIterator.hasNext()) {
     Order order = orderIterator.next();
     // Verify that the order wasn't updated by someone else.
     // Since the query used an index, there was no lock on the row.
     if(order != null && order.date.getTime() >= cancelTime.getTime()) {
       for(OrderLine line : order.lines) {
        // Add the item back to the inventory.
        line.item.quantityOnHand += line.quantity;
        line.quantity = 0;
       }
       em.remove(order);
     }
   }
  em.getTransaction().commit();
}
```

Like the previous example, the cancelOrdersUsingQuery method intends to cancel all orders that were submitted in the past minute. To cancel the order, you find the order using a query, add the items in the order back into the inventory, and remove the order and associated line items from the system.

ObjectQuery tutorial

With the following steps, you can develop a local in-memory ObjectGrid that can store order information for a Web site, and demonstrate how to use ObjectQuery to query the data in the grid.

Before you begin

Be sure to have `objectgrid.jar` file in the classpath.

About this task

Each step in the tutorial builds on the previous step. Follow each of the steps to build a simple Java Platform, Standard Edition Version 1.4 (or later) application that uses an in-memory, local ObjectGrid.

Procedure

1. "ObjectQuery tutorial - step 1"
   - How to create a local ObjectGrid
   - How to define a schema for a single object using field-access
   - How to store the object
   - How to query the object with ObjectQuery
2. "ObjectQuery tutorial - step 2" on page 177
   - How to create an index that the query can use
3. "ObjectQuery tutorial - step 3" on page 178
   - How to create a schema with two related entities
   - How to store objects with a foreign-key reference between them
   - How to query the objects using a single query with a JOIN
4. "ObjectQuery tutorial - step 4" on page 180
   - How to create a schema with multiple related entities
   - How to use method or property access instead of field access

ObjectQuery tutorial - step 1

With the following steps, you can continue to develop a local, in-memory ObjectGrid that stores order information for an online retail store using the ObjectMap APIs. You define a schema for the map and run a query against the map.

Procedure

1. Create an ObjectGrid with a map schema.

   Create an ObjectGrid with one map schema for the map, then insert an object into the cache and later retrieve it using a simple query.

   `OrderBean.java`

   ```
   public class OrderBean implements Serializable {
       String orderNumber;
       java.util.Date date;
       String customerName;
       String itemName;
       int quantity;
       double price;
   }
   ```
2. Define the primary key.

   The previous code shows an OrderBean object. This object implements the java.io.Serializable interface because all objects in the cache must (by default) be Serializable.

The orderNumber attribute is the primary key of the object. The following example program can be run in stand-alone mode. You should follow this tutorial in an Eclipse Java project that has the `objectgrid.jar` file added to the class path.

Application.java

```
package querytutorial.basic.step1;

import java.util.Iterator;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.config.QueryConfig;
import com.ibm.websphere.objectgrid.config.QueryMapping;
import com.ibm.websphere.objectgrid.query.ObjectQuery;

public class Application
{

    static public void main(String [] args) throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping("Order", OrderBean.class.getName(),
     "orderNumber", QueryMapping.FIELD_ACCESS));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");

        s.begin();
        OrderBean o = new OrderBean();
        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.put(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery("SELECT o FROM Order o WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        o = (OrderBean) result.next();
        System.out.println("Found order for customer: " + o.customerName);
        s.commit();
    }
}
```

This eXtreme Scale application first initializes a local ObjectGrid with an automatically generated name. Next, the application creates a BackingMap and a QueryConfig that defines what Java type is associated with the map, the name of the field that is the primary key for the map, and how to access the data in the object. You then obtain a Session to get the ObjectMap instance and insert an OrderBean object into the map in a transaction.

After the data is committed into the cache, you can use ObjectQuery to find the OrderBean using any of the persistent fields in the class. Persistent fields are those that do not have the transient modifier. Because you did not define any indexes on the BackingMap, ObjectQuery must scan each object in the map using Java reflection.

What to do next

"ObjectQuery tutorial - step 2" demonstrates how an index can be used to optimize the query.

ObjectQuery tutorial - step 2

With the following steps, you can continue to create an ObjectGrid with one map and an index, along with a schema for the map. Then you can insert an object into the cache and later retrieve it using a simple query.

Before you begin

Be sure that you have completed "ObjectQuery tutorial - step 1" on page 176 before proceeding with this step of the tutorial.

Procedure

Schema and index

Application.java

```
// Create an index
    HashIndex idx= new HashIndex();
    idx.setName("theItemName");
    idx.setAttributeName("itemName");
    idx.setRangeIndex(true);
    idx.setFieldAccessAttribute(true);
    orderBMap.addMapIndexPlugin(idx);
}
```

The index must be a com.ibm.websphere.objectgrid.plugins.index.HashIndex instance with the following settings:

- The Name is arbitrary, but must be unique for a given BackingMap.
- The AttributeName is the name of the field or bean property which the indexing engine uses to introspect the class. In this case, it is the name of the field for which you will create an index.
- RangeIndex must always be true.
- FieldAccessAttribute should match the value set in the QueryMapping object when the query schema was created. In this case, the Java object is accessed using the fields directly.

When a query runs that filters on the itemName field, the query engine automatically uses the defined index. Using the index allows the query to run much faster and a map scan is not needed. The next step demonstrates how an index can be used to optimize the query.
Next step

ObjectQuery tutorial - step 3

With the following step, you can create an ObjectGrid with two maps and a schema for the maps with a relationship, then insert objects into the cache and later retrieve them using a simple query.

Before you begin

Be sure you have completed "ObjectQuery tutorial - step 2" on page 177 prior to proceeding with this step.

About this task

In this example, there are two maps, each with a single Java type mapped to it. The Order map has OrderBean objects and the Customer map has CustomerBean objects in it.

Procedure

Define maps with a relationship.

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    int quantity;
    double price;
}
```

The OrderBean no longer has the customerName in it. Instead, it has the customerId, which is the primary key for the CustomerBean object and the Customer map.

CustomerBean.java

```
public class CustomerBean implements Serializable{
    private static final long serialVersionUID = 1L;
    String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

The relationship between the two types or Maps follows:

Application.java

```
public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");
        og.defineMap("Customer");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping(
            "Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryMapping(new QueryMapping(
            "Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryRelationship(new QueryRelationship(
             OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");
        ObjectMap custMap = s.getMap("Customer");

        s.begin();
        CustomerBean cust = new CustomerBean();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        custMap.insert(cust.id, cust);

        OrderBean o = new OrderBean();
        o.customerId = cust.id;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.insert(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery(
            "SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        cust = (CustomerBean) result.next();
        System.out.println("Found order for customer: " + cust.firstName + " " + cust.surname);
        s.commit();
    }
}
```

The equivalent XML in the ObjectGrid deployment descriptor 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="CompanyGrid">
      <backingMap name="Order"/>
      <backingMap name="Customer"/>

      <querySchema>
         <mapSchemas>
           <mapSchema
             mapName="Order"
             valueClass="com.mycompany.OrderBean"
             primaryKeyField="orderNumber"
             accessType="FIELD"/>
           <mapSchema
             mapName="Customer"
             valueClass="com.mycompany.CustomerBean"
             primaryKeyField="id"
             accessType="FIELD"/>
         </mapSchemas>
         <relationships>
           <relationship
             source="com.mycompany.OrderBean"
             target="com.mycompany.CustomerBean"
             relationField="customerId"/>
         </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

What to do next

"ObjectQuery tutorial - step 4," expands the current step by including field and property access objects and additional relationships.

ObjectQuery tutorial - step 4

The following step shows how to create an ObjectGrid with four maps and a schema for the maps with multiple uni-directional and bi-directional relationships. Then you can insert objects into the cache and later retrieve them using several queries.

Before you begin

Be sure to have completed "ObjectQuery tutorial - step 3" on page 178 prior to continuing with the current step.

Procedure

Multiple map relationships

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    int quantity;
    double price;
}
```

As in the previous step, OrderBean no longer has the customerName in it. Instead, it has the customerId, which is the primary key for the CustomerBean object and the Customer map.

CustomerBean.java

```
public class CustomerBean implements Serializable{
    private static final long serialVersionUID = 1L;
    String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

Having created the classes specified above, you may run the application below.

Application.java

```
public class Application
{

    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");
        og.defineMap("Customer");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping(
            "Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryMapping(new QueryMapping(
            "Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryRelationship(new QueryRelationship(
             OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");
        ObjectMap custMap = s.getMap("Customer");

        s.begin();
        CustomerBean cust = new CustomerBean();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        custMap.insert(cust.id, cust);

        OrderBean o = new OrderBean();
        o.customerId = cust.id;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.insert(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery(
            "SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        cust = (CustomerBean) result.next();
        System.out.println("Found order for customer: " + cust.firstName + " " + cust.surname);
        s.commit();
    }
}
```

Using the XML configuration below (in the ObjectGrid deployment descriptor) is equivalent to the programmatic approach above.

```
<?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="og1">
      <backingMap name="Order"/>
      <backingMap name="Customer"/>

      <querySchema>
         <mapSchemas>
           <mapSchema
             mapName="Order"
             valueClass="com.mycompany.OrderBean"
             primaryKeyField="orderNumber"
             accessType="FIELD"/>
           <mapSchema
             mapName="Customer"
             valueClass="com.mycompany.CustomerBean"
             primaryKeyField="id"
             accessType="FIELD"/>
         </mapSchemas>
         <relationships>
           <relationship
             source="com.mycompany.OrderBean"
             target="com.mycompany.CustomerBean"
             relationField="customerId"/>
         </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Java SE security tutorial: overview

With the following tutorial, you can create a distributed eXtreme Scale environment in a Java Platform, Standard Edition environment.

Before you begin

Ensure that you are familiar with the basics of a distributed eXtreme Scale configuration.

About this task

In this tutorial, the catalog server, container server, and client all run in a Java SE environment. Each step in the tutorial builds on the previous one. Follow each of the steps to secure a distributed eXtreme Scale and develop a simple Java SE application to access the secured eXtreme Scale.

Begin tutorial

Procedure

1. "Java SE security tutorial - Step 1" on page 183
   - Start an unsecured catalog server
   - Start an unsecured container server
   - Start a client to access the data
   - Use xsadmin to show map size

  - Stop server
2. "Java SE security tutorial - Step 2" on page 186
   - Use of CredentialGenerator
   - Use of Authenticator
   - Start a secure catalog server
   - Start a secure container server
   - Start client to access secured ObjectGrid
   - Use xsadmin to show map size
   - Stop secure server
3. "Java SE security tutorial - Step 3" on page 192
   - Use of JAAS authorization policy
4. "Java SE security tutorial - Step 4" on page 196
   - Create a key store and trust store
   - Configure SSL properties for the server
   - Configure SSL properties for the client
   - Use xsadmin to show map size
   - Stop secure server

Java SE security tutorial - Step 1

This topic describes a *simple unsecured sample*. Additional security features are added incrementally in the steps of the tutorial to increase the amount of integrated security that is available.

Before you begin

Note: All of the files required for this step of the tutorial are provided in the following section.

Procedure

Running the sample
Start the catalog service by using the following scripts. For more information about starting the catalog service, see the information about starting the catalog service in the *Administration Guide*.

1. Navigate to the bin directory: `cd objectgridRoot/bin`
2. Start a catalog server named catalogServer:
   - UNIX Linux `startOgServer.sh catalogServer`
   - Windows `startOgServer.bat catalogServer`
3. Navigate to the bin directory `cd objectgridRoot/bin`
4. Then launch a container server named c0 with the following script:
   - UNIX Linux
     ```
     startOgServer.sh c0 -objectGridFile ../xml/SimpleApp.xml -deploymentPolicyFile ../xml/SimpleDP.xml
     -catalogServiceEndPoints localhost:2809
     ```
   - Windows
     ```
     startOgServer.bat c0 -objectGridFile ../xml/SimpleApp.xml - deploymentPolicyFile ../xml/SimpleDP.xml
     -catalogServiceEndPoints localhost:2809
     ```

Example

For more information about starting container servers, see the information about starting the container processes in the *Administration Guide*.

After the catalog server and container server have been started, launch the client as follows.

1. Navigate to the bin directory one more time.
2.
```
java -classpath ../lib/objectgrid.jar;../applib/secsample.jar
com.ibm.websphere.objectgrid.security.sample.guide.SimpleApp
```

The `secsample.jar` file contains the SimpleApp class.

The output of this program is:

```
The customer name for ID 0001 is fName lName
```

You may also use xsadmin to show the mapsizes of the "accounting" grid.

- Navigate to the directory `objectgridRoot/bin`.
- Use the `xsadmin` command with option -mapSizes as follows.
  - UNIX Linux `xsadmin.sh -g accounting -m mapSet1 -mapSizes`
  - Windows `xsadmin.bat -g accounting -m mapSet1 -mapSizes`

  You will see the following output.

```
This administrative utility is provided as a sample only and is not to be
considered a fully supported component of the WebSphere eXtreme Scale
product.
Connecting to Catalog service at localhost:1099
*********** Displaying Results for Grid - accounting, MapSet - mapSet1
***********
*** Listing Maps for c0 ***
Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary
Server Total: 1
Total Domain Count: 1
```

Stopping servers

Container server

Use the following command to stop the container server c0.

UNIX Linux `stopOgServer.sh c0 -catalogServiceEndPoints localhost:2809`

Windows `stopOgServer.bat c0 -catalogServiceEndPoints localhost:2809`

You will see the following message.

```
CWOBJ2512I: ObjectGrid server c0 stopped.
```

Catalog server

You can stop a catalog server using the following command.

UNIX Linux `stopOgServer.sh catalogServer -catalogServiceEndPoints localhost:2809`

Windows `stopOgServer.bat catalogServer -catalogServiceEndPoints localhost:2809`

If you shut down the catalog server, you will see the following message.

```
CWOBJ2512I: ObjectGrid server catalogServer stopped.
```

Required files

The file below is the Java class for SimpleApp.

SimpleApp.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 (C) COPYRIGHT International Business Machines Corp. 2007-2009
package com.ibm.websphere.objectgrid.security.sample.guide;

import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;

public class SimpleApp {

    public static void main(String[] args) throws Exception {

        SimpleApp app = new SimpleApp();
        app.run(args);
    }

    /**
     * read and write the map
     * @throws Exception
     */
    protected void run(String[] args) throws Exception {
        ObjectGrid og = getObjectGrid(args);

        Session session = og.getSession();

        ObjectMap customerMap = session.getMap("customer");

        String customer = (String) customerMap.get("0001");

        if (customer == null) {
            customerMap.insert("0001", "fName lName");
        } else {
            customerMap.update("0001", "fName lName");
        }
        customer = (String) customerMap.get("0001");

        System.out.println("The customer name for ID 0001 is " + customer);
    }

    /**
     * Get the ObjectGrid
     * @return an ObjectGrid instance
     * @throws Exception
     */
    protected ObjectGrid getObjectGrid(String[] args) throws Exception {
        ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

        // Create an ObjectGrid
        ClientClusterContext ccContext = ogManager.connect("localhost:2809", null, null);
        ObjectGrid og = ogManager.getObjectGrid(ccContext, "accounting");
        return og;

    }

}
```

The getObjectGrid method in this class obtains an ObjectGrid, and the run method reads a record from the customer map and updates the value.

To run this sample in a distributed environment, an ObjectGrid descriptor XML file `SimpleApp.xml`, and a deployment XML file, `SimpleDP.xml`, are created. The files are featured in the following example:

`SimpleApp.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="accounting">
            <backingMap name="customer" readOnly="false" copyKey="true"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

The following XML file configures the deployment environment.

`SimpleDP.xml`

```
<?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="accounting">
  <mapSet name="mapSet1" numberOfPartitions="1" minSyncReplicas="0" maxSyncReplicas="2"
   maxAsyncReplicas="1">
   <map ref="customer"/>
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

This is a simple ObjectGrid configuration with one ObjectGrid instance named "accounting" and one map named "customer" (within the mapSet "mapSet1"). The `SimpleDP.xml` file features one map set that is configured with 1 partition and 0 minimum required replicas.

Next step of tutorial

Java SE security tutorial - Step 2

Building on the previous step, the following topic shows how to implement client authentication in a distributed eXtreme Scale environment.

Before you begin

Be sure that you have completed “Java SE security tutorial - Step 1” on page 183.

About this task

With client authentication enabled, a client is authenticated before connecting to the eXtreme Scale server. This section demonstrates how client authentication can be done in an eXtreme Scale server environment, including sample code and scripts to demonstrate.

As any other authentication mechanism, the minimum authentication consists of the following steps:

1. The administrator changes configurations to make authentication a requirement.
2. The client provides a credential to the server.
3. The server authenticates the credential to the registry.

Procedure

1. **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. Refer to the Credential 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.

   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 used to generate a client credential. This is useful when the credential is expirable. In this case, the getCredential() method is called to renew a credential. Refer to CredentialGenerator API Documentation for more details.

   You can implement these two interfaces for eXtreme Scale client runtime to obtain client credentials.

   This sample uses the following two sample plug-in implementations provided by eXtreme Scale.

   ```
   com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential
   ```

   ```
   com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator
   ```

   For more information about these plug-ins, see the topic on client authentication programming in the *Programming Guide*.

2. **Server authenticator** 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 eXtreme Scale 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. Refer to Authenticator API Documentation for more details.

   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 production). client authentication programming in the *Programming Guide*.

   This KeyStoreLoginAuthenticator uses a KeyStoreLoginModule to authenticate the user with the key store by using the JAAS login module "KeyStoreLogin". The key store can be configured as an option to the KeyStoreLoginModule class. The following example illustrates the keyStoreLogin alias configured in the JAAS configuration file og_jaas.config:

```
KeyStoreLogin{
com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginModule required
     keyStoreFile="../security/sampleKS.jks" debug = true;
};
```

The following commands create a key store sampleKS.jks in the %OBJECTGRID_HOME%/security directory with the password as sampleKS1. Also, three user certificates representing the administrator user, the manager user, and the cashier user are created with their own passwords.

a. Navigate to the eXtreme Scale root directory.
```
cd objectgridRoot
```
b. Create a directory called "security".
```
mkdir security
```
c. Navigate to the newly created security directory.
```
cd security
```
d. Use keytool (in the `javaHOME/bin` directory) to create a user "administator" with password "administrator1" in the key store sampleKS.jks.
```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias administrator -keypass administrator1
-dname CN=administrator,O=acme,OU=OGSample -validity 10000
```
e. Use keytool (in the `javaHOME/bin` directory) to create a user "manager" with password "manager1" in the key store sampleKS.jks.
```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias manager -keypass manager1
-dname CN=manager,O=acme,OU=OGSample -validity 10000
```
f. Use keytool (in the `javaHOME/bin` directory) to create a user "cashier" with password "cashier1" in the key store sampleKS.jks.
```
keytool -genkey -v -keystore ./sampleKS.jks -storepass sampleKS1
-alias cashier -keypass cashier1 -dname CN=cashier,O=acme,OU=OGSample
-validity 10000
```

The client security configuration is configured in the client properties file. Use the following command to create a copy in the `%OBJECTGRID_HOME%/security` directory:

a. Change to the security directory.
```
cd objectgridRoot/security
```
b. Copy the sampleClient.properties file to the client.properties file.
```
cp ../properties/sampleClient.properties client.properties
```

The following properties are highlighted in the client.properties file in the security directory.

a. **securityEnabled:** Setting securityEnabled to true (default value) enables the client security, which includes authentication.
b. **credentialAuthentication:** Set credentialAuthentication to Supported (default value), which means the client supports credential authentication.
c. **transportType:** Set transportType to TCP/IP, which means no SSL will be used.
d. **singleSignOnEnabled:** Set it to false (default value). Single sign-on is not available.

3. **Server security configuration**

The server security configuration is specified in the security descriptor XML file and the server security property file.The security descriptor XML file describes the security properties common to all servers (including catalog servers and container servers). One property example is the authenticator configuration which represents the user registry and authentication mechanism.

Here is the `security.xml` file to be used in this sample:

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

 <security securityEnabled="true" loginSessionExpirationTime="300" >

        <authenticator className ="com.ibm.websphere.objectgrid.security.plugins.builtins.
     KeyStoreLoginAuthenticator">
        </authenticator>
    </security>

</securityConfig>
```

a. **securityEnabled:** Set to true, which enables the server security including authentication.

b. **loginSessionExpirationTime:** Set the value to 300 (default value).

c. **authenticator:** Add the authenticator class KeyStoreLoginAuthenticator to the cluster XML file as follows:

```
<authenticator className ="com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator">
        </authenticator>
```

d. **credentialAuthentication:** Set credentialAuthentication attribute to Required so the server requires authentication

For more detailed explanation on the `security.xml` file, see the information about the security descriptor XML file in the *Administration Guide*.

Copy the server properties file into the security directory. At this time, you do not need to modify anything in this file.

a. Navigate to the security directory.

```
cd objectgridRoot/security
```

b. Copy the sample objectGrid `sampleServer.properties` file from the properties directory to the new `server.properties` file.

```
cp ../properties/containerServer.properties server.properties
```

Make the following changes in the `server.properties` file:

a. **securityEnabled:** Set the **`securityEnabled`** attribute to true.

b. **transportType:** Set **`transportType`** attribute to TCP/IP, which means no SSL is used.

c. **secureTokenManagerType:** Set **`secureTokenManagerType`** attribute to none to not configure the secure token manager.

4. **Secure client** Connect the client application to the server securely as demonstrated in the following example:

```
package com.ibm.websphere.objectgrid.security.sample.guide;

import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration;
import com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory;
import com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator;
import com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator;

public class SecureSimpleApp extends SimpleApp {

    public static void main(String[] args) throws Exception {

        SecureSimpleApp app = new SecureSimpleApp();
        app.run(args);
    }

    /**
     * Get the ObjectGrid
     * @return an ObjectGrid instance
     * @throws Exception
     */
    protected ObjectGrid getObjectGrid(String[] args) throws Exception {
        ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
        ogManager.setTraceFileName("logs/client.log");
        ogManager.setTraceSpecification("ObjectGrid*=all=enabled:ORBRas=all=enabled");

        // Creates a ClientSecurityConfiguration object using the specified file
        ClientSecurityConfiguration clientSC = ClientSecurityConfigurationFactory
                .getClientSecurityConfiguration(args[0]);

        // Creates a CredentialGenerator using the passed-in user and password.
        CredentialGenerator credGen = new UserPasswordCredentialGenerator(args[1], args[2]);
        clientSC.setCredentialGenerator(credGen);

        // Create an ObjectGrid by connecting to the catalog server
        ClientClusterContext ccContext = ogManager.connect("localhost:2809", clientSC, null);
        ObjectGrid og = ogManager.getObjectGrid(ccContext, "accounting");

        return og;

    }

}
```

There are three things different from the non-secured application:

a. Created a ClientSecurityConfiguration object by passing the configured `client.properties` file.
b. Created a UserPasswordCredentialGenerator by using the passed-in user ID and password.
c. Connected to the catalog server to obtain an ObjectGrid from the ClientClusterContext by passing a ClientSecurityConfiguration object.

5. **Issue the application**

   To run the application, start the catalog server. Issue the -clusterFile and -serverProps command line options to pass in the security properties:

   a. Navigate to the bin directory:

      `cd objectgridRoot/bin`

   b. Launch the catalog server:

      - UNIX Linux

        ```
        startOgServer.sh catalogServer -clusterSecurityFile ../security/security.xml
        -serverProps ../security/server.properties -jvmArgs
        -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

      - Windows

        ```
        startOgServer.bat catalogServer -clusterSecurityFile ../security/security.xml
        -serverProps ../security/server.properties -jvmArgs
        -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

   Then, launch a secure container server by using the following script:

   a. Navigate to the bin directory again:

      `cd objectgridRoot/bin`

   b. Launch a secure container server:

      - Linux UNIX

        ```
        startOgServer.sh c0 -objectgridFile ../xml/SimpleApp.xml
        -deploymentPolicyFile ../xml/SimpleDP.xml
        -catalogServiceEndPoints localhost:2809
        -serverProps ../security/server.properties
        -jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

      - Windows

        ```
        startOgServer.bat c0 -objectgridFile ../xml/SimpleApp.xml
        -deploymentPolicyFile ../xml/SimpleDP.xml
        -catalogServiceEndPoints localhost:2809
        -serverProps ../security/server.properties
        -jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

The server property file is passed by issuing -serverProps.

After the server is started, start the client by using the following command:

a. `cd objectgridRoot/bin`

b.

```
java -classpath ../lib/objectgrid.jar;../applib/secsample.jar
  com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp
  ../security/client.properties manager manager1
```

Linux Use a colon (:) for the classpath separator instead of a semicolon (;) as in the previous example.

The `secsample.jar` file contains the SimpleApp class.

The SecureSimpleApp uses three parameters that are provided in the following list:

a. The `../security/client.properties` file is the client security property file.

b. `manager` is the user ID.

c. `manager1` is the password.

After you issue the class, the following output results:

The customer name for `ID 0001 is fName lName`.

You may also use xsadmin to show the mapsizes of the "accounting" grid.

- Navigate to the directory `objectgridRoot/bin`.
- Use the `xsadmin` command with option -mapSizes as follows.
  - UNIX Linux `xsadmin.sh -g accounting -m mapSet1 -username manager -password manager1 -mapSizes`
  - Windows `xsadmin.bat -g accounting -m mapSet1 -username manager -password manager1 -mapSizes`

  You see the following output.

```
This administrative utility is provided as a sample only and is not to
be considered a fully supported component of the WebSphere eXtreme
Scale product.

Connecting to Catalog service at localhost:1099

*********** Displaying Results for Grid - accounting, MapSet - mapSet1
***********

*** Listing Maps for c0 ***

Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary

Server Total: 1

Total Domain Count: 1
```

Now you can use stopOgServer command to stop the container server or catalog service process. However you need to provide a security configuration file. The sample client property file defines the following two properties to generate a userID/password credential (manager/manager1).

```
credentialGeneratorClass=com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator
credentialGeneratorProps=manager manager1
```

Stop the container c0 with the following command.

- UNIX Linux `stopOgServer.sh c0 -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..\security\client.properties`
- Windows `stopOgServer.bat c0 -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..\security\client.properties`

If you do not provide the -clientSecurityFile option, you will see an exception with the following message.

```
>> SERVER (id=39132c79, host=9.10.86.47) TRACE START:
```

```
>> org.omg.CORBA.NO_PERMISSION: Server requires credential
authentication but there is no security context from the client. This
usually happens when the client does not pass a credential the server.
vmcid: 0x0
minor code: 0
completed: No
```

You can also shut down the catalog server using the following command. However, if you want to continue trying the next step tutorial, you can let the catalog server stay running.

- UNIX Linux `stopOgServer.sh catalogServer -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..\security\client.properties`
- Windows `stopOgServer.bat catalogServer -catalogServiceEndPoints localhost:2809 -clientSecurityFile ..\security\client.properties`

If you do shutdown the catalog server, you will see the following output.

```
CWOBJ2512I: ObjectGrid server catalogServer stopped
```

Now, you have successfully made your system partially secure by enabling authentication. You configured the server to plug in the user registry, configured the client to provide client credentials, and changed the client property file and cluster XML file to enable authentication.

If you provide an invalidate password, you see an exception stating that the user name or password is not correct.

For more details about client authentication, see the information about application client authentication in the *Administration Guide*.

Next step of tutorial

Java SE security tutorial - Step 3

After authenticating a client, as in the previous step, you can give security privileges through eXtreme Scale authorization mechanisms.

Before you begin

Be sure to have completed “Java SE security tutorial - Step 2” on page 186 prior to proceeding with this task.

About this task

The previous step of this tutorial demonstrated how to enable authentication in an eXtreme Scale grid. As a result, no unauthenticated client can connect to your server and submit requests to your system. However, every authenticated client has the same permission or privileges to the server, such as reading, writing, or deleting data that is stored in the ObjectGrid maps. Clients can also issue any type of query. This section demonstrates how to use eXtreme Scale authorization to give various authenticated users varying privileges.

Similar to many other systems, eXtreme Scale adopts a permission-based authorization mechanism. WebSphere eXtreme Scale has different permission categories that are represented by different permission classes. This topic features MapPermission. For complete category of permissions, see the client authorization reference in the *Programming Guide*..

In WebSphere eXtreme Scale, the com.ibm.websphere.objectgrid.security.MapPermission class represents permissions to the eXtreme Scale 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:

- read: Grants permission to read the data from the map.
- write: Grants permission to update the data in the map.
- insert: Grants permission to insert the data into the map.
- remove: Grants permission to remove the data from the map.
- invalidate: Grants permission to invalidate the data from the map.
- all: Grants all permissions to read, write, insert, remote, and invalidate.

The authorization occurs when a client calls a method of ObjectMap or JavaMap. The eXtreme Scale runtime checks different map permissions for different methods. If the required permissions are not granted to the client, an AccessControlException results.

This tutorial demonstrates how to use Java Authentication and Authorization Service (JAAS) authorization to grant authorization map accesses for different users.

Procedure

1. **Enable eXtreme Scale authorization**. To enable authorization on the ObjectGrid, you need to set the securityEnabled attribute to `true` for that particular ObjectGrid in the XML file. Enabling security on the ObjectGrid means that you are enabling authorization. Use the following commands to create a new ObjectGrid XML file with security enabled.
   a. Navigate to the `xml` directory.
   ```
   cd objectgridRoot/xml
   ```
   b. Copy the `SimpleApp.xml` file to the `SecureSimpleApp.xml` file.
   ```
   cp SimpleApp.xml SecureSimpleApp.xml
   ```
   c. Open the `SecureSimpleApp.xml` file and add `securityEnabled="true"` on the ObjectGrid level as the following XML shows:
   ```
   <?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" securityEnabled="true">
               <backingMap name="customer" readOnly="false" copyKey="true"/>
           </objectGrid>
       </objectGrids>
   </objectGridConfig>
   ```
2. **Define the authorization policy.** In the pre-client authentication section, you created three users in the key store: cashier, manager, and administrator. In this example, the user "cashier" only has read permissions to all the maps, and the user "manager" has all permissions. JAAS authorization is used in this example. JAAS authorization uses authorization policy file to grant permissions to principals. The following file is defined in the security directory:
   ```
   grant codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
       principal javax.security.auth.x500.X500Principal "CN=cashier,O=acme,OU=OGSample" {
       permission com.ibm.websphere.objectgrid.security.MapPermission "accounting.*", "read ";
   };

   grant codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
       principal javax.security.auth.x500.X500Principal "CN=manager,O=acme,OU=OGSample" {
       permission com.ibm.websphere.objectgrid.security.MapPermission "accounting.*", "all";
   };
   ```

Note:

- The codebase `"http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"` is a specially-reserved URL for ObjectGrid. All ObjectGrid permissions granted to principals should use this special code base.
- The first grant statement grants "read" map permission to principal `"CN=cashier,O=acme,OU=OGSample"`, so the cashier has only map read permission to all the maps in the ObjectGrid accounting.
- The second grant statement grants "all" map permission to principal `"CN=manager,O=acme,OU=OGSample"`, so the manager has all permissions to maps in the ObjectGrid accounting.

Now you can launch a server with an authorization policy. The JAAS authorization policy file can be set using the standard -D property: -Djava.security.auth.policy=../security/ogAuth.policy

3. **Run the application.**

   After you create the above files, you can run the application.

   Use the following commands to start the catalog server. For more information about starting the catalog service, see the information about starting a catalog service in the *Administration Guide*.

   a. Navigate to the bin directory: `cd objectgridRoot/bin`

   b. Start the catalog server.

   - UNIX Linux
     ```
     startOgServer.sh catalogServer
     -clusterSecurityFile ../security/security.xml -serverProps
     ../security/server.properties -jvmArgs
     -Djava.security.auth.login.config="../security/og_jaas.config"
     ```
   - Windows
     ```
     startOgServer.bat catalogServer -clusterSecurityFile
     ../security/security.xml -serverProps ../security/server.properties
     -jvmArgs -Djava.security.auth.login.config="../security/
     og_jaas.config"
     ```

   The `security.xml` and `server.properties` files were created in the previous step of this tutorial.

   T

   c. You can then start a secure container server using the following script. Run the following script from the bin directory:

   - UNIX Linux
     ```
     # startOgServer.sh c0 -objectGridFile
     ../xml/SecureSimpleApp.xml -deploymentPolicyFile
     ../xml/SimpleDP.xml -catalogServiceEndPoints localhost:2809
     -serverProps ../security/server.properties -jvmArgs
     -Djava.security.auth.login.config="../security/og_jaas.config"
     -Djava.security.auth.policy="../security/og_auth.policy"
     ```
   - Windows
     ```
     startOgServer.bat c0 -objectGridFile ../xml/
     SecureSimpleApp.xml -deploymentPolicyFile ../xml/SimpleDP.xml
     -catalogServiceEndPoints localhost:2809 -serverProps
     ../security/server.properties -jvmArgs
     -Djava.security.auth.login.config="../security/og_jaas.config"
     -Djava.security.auth.policy="../security/og_auth.policy"
     ```

   Notice the following differences from the previous container server start command:

   - Use the `SecureSimpleApp.xml` file instead of the `SimpleApp.xml` file.
   - Add another `-Djava.security.auth.policy` argument to set the JAAS authorization policy file to the container server process.

Use the same command as in the previous step of the tutorial:

a. Navigate to the bin directory.

b. `java -classpath ../lib/objectgrid.jar;../applib/secsample.jar com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp ../security/client.properties manager manager1`

   Because user "manager" has all permissions to maps in the accounting ObjectGrid, the application runs properly.

   Now, instead of using user "manager", use user "cashier" to launch the client application.

c. Navigate to the bin directory.

d. `java -classpath ../lib/objectgrid.jar;../applib/secsample.jar com.ibm.ws.objectgrid.security.sample.guide.SecureSimpleApp ../security/client.properties cashier cashier1`

The following exception results:

```
Exception in thread "P=387313:O=0:CT" com.ibm.websphere.objectgrid.TransactionException:
rolling back transaction, see caused by exception
 at com.ibm.ws.objectgrid.SessionImpl.rollbackPMapChanges(SessionImpl.java:1422)
  at com.ibm.ws.objectgrid.SessionImpl.commit(SessionImpl.java:1149)
  at com.ibm.ws.objectgrid.SessionImpl.mapPostInvoke(SessionImpl.java:2260)
  at com.ibm.ws.objectgrid.ObjectMapImpl.update(ObjectMapImpl.java:1062)
  at com.ibm.ws.objectgrid.security.sample.guide.SimpleApp.run(SimpleApp.java:42)
 at com.ibm.ws.objectgrid.security.sample.guide.SecureSimpleApp.main(SecureSimpleApp.java:27)
Caused by: com.ibm.websphere.objectgrid.ClientServerTransactionCallbackException:
   Client Services - received exception from remote server:
     com.ibm.websphere.objectgrid.TransactionException: transaction rolled back,
   see caused by Throwable
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.processReadWriteResponse(
            RemoteTransactionCallbackImpl.java:1399)
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.processReadWriteRequestAndResponse(
            RemoteTransactionCallbackImpl.java:2333)
        at com.ibm.ws.objectgrid.client.RemoteTransactionCallbackImpl.commit(RemoteTransactionCallbackImpl.java:557)
        at com.ibm.ws.objectgrid.SessionImpl.commit(SessionImpl.java:1079)
        ... 4 more
Caused by: com.ibm.websphere.objectgrid.TransactionException: transaction rolled back, see caused by Throwable
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processLogSequence(ServerCoreEventProcessor.java:1133)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processReadWriteTransactionRequest
     (ServerCoreEventProcessor.java:910)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processClientServerRequest(ServerCoreEventProcessor.java:1285)

        at com.ibm.ws.objectgrid.ShardImpl.processMessage(ShardImpl.java:515)
        at com.ibm.ws.objectgrid.partition.IDLShardPOA._invoke(IDLShardPOA.java:154)
        at com.ibm.CORBA.poa.POAServerDelegate.dispatchToServant(POAServerDelegate.java:396)
        at com.ibm.CORBA.poa.POAServerDelegate.internalDispatch(POAServerDelegate.java:331)
        at com.ibm.CORBA.poa.POAServerDelegate.dispatch(POAServerDelegate.java:253)
        at com.ibm.rmi.iiop.ORB.process(ORB.java:503)
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:1553)
        at com.ibm.rmi.iiop.Connection.respondTo(Connection.java:2680)
        at com.ibm.rmi.iiop.Connection.doWork(Connection.java:2554)
        at com.ibm.rmi.iiop.WorkUnitImpl.doWork(WorkUnitImpl.java:62)
        at com.ibm.rmi.iiop.WorkerThread.run(ThreadPoolImpl.java:202)
        at java.lang.Thread.run(Thread.java:803)
Caused by: java.security.AccessControlException: Access denied (
   com.ibm.websphere.objectgrid.security.MapPermission accounting.customer write)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:155)
        at com.ibm.ws.objectgrid.security.MapPermissionCheckAction.run(MapPermissionCheckAction.java:141)
        at java.security.AccessController.doPrivileged(AccessController.java:275)
        at javax.security.auth.Subject.doAsPrivileged(Subject.java:727)
        at com.ibm.ws.objectgrid.security.MapAuthorizer$1.run(MapAuthorizer.java:76)
        at java.security.AccessController.doPrivileged(AccessController.java:242)
        at com.ibm.ws.objectgrid.security.MapAuthorizer.check(MapAuthorizer.java:66)
        at com.ibm.ws.objectgrid.security.SecuredObjectMapImpl.checkMapAuthorization(SecuredObjectMapImpl.java:429)
        at com.ibm.ws.objectgrid.security.SecuredObjectMapImpl.update(SecuredObjectMapImpl.java:490)
        at com.ibm.ws.objectgrid.SessionImpl.processLogSequence(SessionImpl.java:1913)
        at com.ibm.ws.objectgrid.SessionImpl.processLogSequence(SessionImpl.java:1805)
        at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processLogSequence(ServerCoreEventProcessor.java:1011)
        ... 14 more
```

This exception occurs because the user "cashier" does not have write permission, so it cannot update the map customer.

Now your system supports authorization. You can define authorization policies to grant different permissions to different users. For more information about authorization, see the information about application client authorization in the *Programming Guide*.

What to do next

Complete the next step of the tutorial. See "Java SE security tutorial - Step 4."

Java SE security tutorial - Step 4

The following step explains how you can enable a security layer for communication between your environment's endpoints.

Before you begin

Be sure you have completed "Java SE security tutorial - Step 3" on page 192 prior to proceeding with this task.

About this task

The eXtreme Scale topology supports both Transport Layer Security/Secure Sockets Layer (TLS/SSL) for secure communication between ObjectGrid endpoints (client, container servers, and catalog servers). This step of the tutorial builds upon the previous steps to enable transport security.

Procedure

1. **Create TLS/SSL keys and key stores**

   In order to enable transport security, you must create a key store and trust store. This exercise only creates one key and trust-store pair. These stores are used for ObjectGrid clients, container servers, and catalog servers, and are created with the JDK keytool.

   - *Create a private key in the key store*

     ```
     keytool -genkey -alias ogsample -keystore key.jks -storetype JKS
     -keyalg rsa -dname "CN=ogsample, OU=Your Organizational Unit, O=Your
     Organization, L=Your City, S=Your State, C=Your Country" -storepass
     ogpass -keypass ogpass -validity 3650
     ```

     Using this command, a key store key.jks is created with a key "ogsample" stored in it. This key store key.jks will be used as the SSL key store.
   - *Export the public certificate*

     ```
     keytool -export -alias ogsample -keystore key.jks -file temp.key
     -storepass ogpass
     ```

     Using this command, the public certificate of key "ogsample" is extracted and stored in the file temp.key.
   - *Import the client's public certificate to the trust store*

     ```
     keytool -import -noprompt -alias ogsamplepublic -keystore trust.jks
     -file temp.key -storepass ogpass
     ```

     Using this command, the public certificate was added to key store trust.jks. This trust.jks is used as the SSL trust store.
2. **Configuring ObjectGrid property files**

   In this step, you must configure the ObjectGrid property files to enable transport security.

First, copy the key.jks and trust.jks files into the objectgridRoot/security directory.

We set the following properties in the client.properties and server.properties file.

```
transportType=SSL-Required

alias=ogsample
contextProvider=IBMJSSE2
protocol=SSL
keyStoreType=JKS
keyStore=../security/key.jks
keyStorePassword=ogpass
trustStoreType=JKS
trustStore=../security/trust.jks
trustStorePassword=ogpass
```

transportType: The value of transportType is set to "SSL-Required", which means the transport requires SSL. So all the ObjectGrid endpoints (clients, catalog servers, and container servers) should have SSL configuration set and all transport communication will be encrypted.

The other properties are used to set the SSL configurations. See the information about transport layer security and the secure sockets layer in the *Administration Guide* for a detailed explanation. Make sure you follow the instructions in this topic to update your orb.properties file.

Make sure you follow this page to update your `orb.properties` file.

In the `server.properties` file, you must add an additional property clientAuthentication and set it to false. On the server side, you do not need to trust the client.

```
clientAuthentication=false
```

3. **Run the application**

   The commands are the same as the commands in the "Java SE security tutorial - Step 3" on page 192 topic.

   Use the following commands to start a catalog server.

   a. Navigate to the bin directory: `cd objectgridRoot/bin`
   b. Start the catalog server:
      - Linux UNIX

        ```
        startOgServer.sh catalogServer -clusterSecurityFile ../security/security.xml
        -serverProps ../security/server.properties -JMXServicePort 11001
        -jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

      - Windows

        ```
        startOgServer.bat catalogServer -clusterSecurityFile ../security/security.xml
        -serverProps ../security/server.properties -JMXServicePort 11001 -jvmArgs
        -Djava.security.auth.login.config="../security/og_jaas.config"
        ```

      The `security.xml` and `server.properties` files were created in the "Java SE security tutorial - Step 2" on page 186 page.

      Use the -JMXServicePort option to explicitly specify the JMX port for the server. This option is required to use the xsadmin command.

      Run a secure ObjectGrid container server:
   c. Navigate to the bin directory again: `cd objectgridRoot/bin`
   d.
      - Linux UNIX

        ```
        startOgServer.sh c0 -objectGridFile ../xml/SecureSimpleApp.xml
        -deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndPoints
        localhost:2809 -serverProps ../security/server.properties
        ```

```
-JMXServicePort 11002 -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

- Windows

  ```
  startOgServer.bat c0 -objectGridFile ../xml/SecureSimpleApp.xml
  -deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndPoints localhost:2809
  -serverProps ../security/server.properties -JMXServicePort 11002
  -jvmArgs -Djava.security.auth.login.config="../security/og_jaas.config"
  -Djava.security.auth.policy="../security/og_auth.policy"
  ```

Notice the following differences from the previous container server start command:

- Use SecureSimpleApp.xml instead of SimpleApp.xml
- Add another -Djava.security.auth.policy to set the JAAS authorization policy file to the container server process.

Run the following command for client authentication:

a. `cd objectgridRoot/bin`
b.

   ```
   javaHome/java -classpath ../lib/objectgrid.jar;../applib/secsample.jar
   com.ibm.websphere.objectgrid.security.sample.guide.SecureSimpleApp
   ../security/client.properties manager manager1
   ```

   Because user "manager" has permission to all the maps in the accounting ObjectGrid, the application runs successfully.

You may also use xsadmin to show the mapsizes of the "accounting" grid.

- Navigate to the directory `objectgridRoot/bin`.
- Use the `xsadmin` command with option -mapSizes as follows.
  - UNIX Linux

    ```
    xsadmin.sh -g accounting -m mapSet1 -mapsizes -p 11001 -ssl
    -trustpath ..\security\trust.jks -trustpass ogpass -trusttype jks
    -username manager -password manager1
    ```

  - Windows

    ```
    xsadmin.bat -g accounting -m mapSet1 -mapsizes -p 11001 -ssl
    -trustpath ..\security\trust.jks -trustpass ogpass -trusttype jks
    -username manager -password manager1
    ```

  Notice we specify the JMX port of the catalog service using -p 11001 here.

  You see the following output.

  ```
  This administrative utility is provided as a sample only and is not to
  be considered a fully supported component of the WebSphere eXtreme Scale product.
  Connecting to Catalog service at localhost:1099
  *********** Displaying Results for Grid - accounting, MapSet - mapSet1 ***********
  *** Listing Maps for c0 ***
  Map Name: customer Partition #: 0 Map Size: 1 Shard Type: Primary
  Server Total: 1
  Total Domain Count: 1
  ```

Running the application with an incorrect key store

If your trust store does not contain the public certificate of the private key in the key store, you will get an exception complaining that the key cannot be trusted.

In order to show this, create another key store key2.jks.

```
keytool -genkey -alias ogsample -keystore key2.jks -storetype JKS
-keyalg rsa -dname "CN=ogsample, OU=Your Organizational Unit, O=Your
Organization, L=Your City, S=Your State, C=Your Country" -storepass
ogpass -keypass ogpass -validity 3650
```

Then modify the server.properties to make the keyStore point to this new key store key2.jks:

```
keyStore=../security/key2.jks
```

Run the following command to start the catalog server:

a. Navigate to bin: `cd objectgridRoot/bin`

b. Start the catalog server:

Linux UNIX

```
startOgServer.sh c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndPoints localhost:2809
-serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

Windows

```
startOgServer.bat c0 -objectGridFile ../xml/SecureSimpleApp.xml
-deploymentPolicyFile ../xml/SimpleDP.xml -catalogServiceEndPoints localhost:2809
 -serverProps ../security/server.properties -jvmArgs
-Djava.security.auth.login.config="../security/og_jaas.config"
-Djava.security.auth.policy="../security/og_auth.policy"
```

You see the following exception:

```
Caused by: com.ibm.websphere.objectgrid.ObjectGridRPCException:
    com.ibm.websphere.objectgrid.ObjectGridRuntimeException:
        SSL connection fails and plain socket cannot be used.
```

Finally, change the `server.properties` file back to use the `key.jks` file.

REST data services sample and tutorial

This topic describes how to quickly get started with the WebSphere eXtreme Scale REST data service. Instructions are provided forWebSphere Application Server version 7.0,WebSphere Application Server Community Edition and Apache Tomcat.

About this task

The included sample has source code and compiled binaries to run a partitioned eXtreme Scale data grid. This sample demonstrates how to create a simple data grid, model the data using eXtreme Scale entities and provides two command-line client applications that allow adding and querying entities using Java or C# (see Figure 1).

The sample Java client uses the eXtreme Scale Java EntityManager API to persist and query data in the grid. This client can be run in Eclipse or using a command-line script. Note that the sample Java client does not demonstrate the REST data service, but allows updating data in the grid, so a web browser or other clients can read the data. The sample Java client and web browser, as shown in Figure 1, illustrate HTTP clients using the REST data service and eXtreme Scale Java clients using the same eXtreme Scale grid and data contained therein.

The sample Microsoft WCF Data Services C# client communicates with the eXtreme Scale data grid through the REST data service using the .NET framework. The WCF Data Services client can be used to both update and query the data grid.

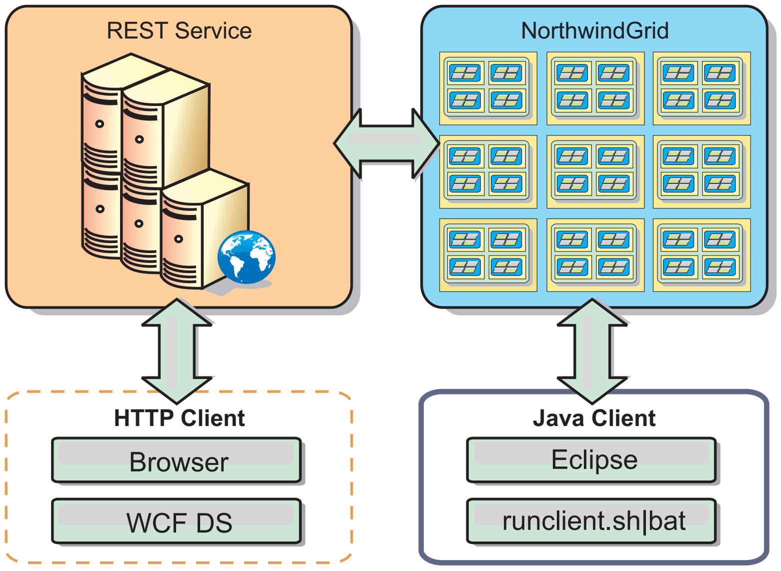


Figure 43. Getting started sample topology

Procedure

1. Configure and start the eXtreme Scale data grid. See "Enabling the REST data service" on page 202.
2. Configure and start the REST data service in a web server. See "Configuring application servers for the REST data service" on page 209.
3. Run a client to interact with the REST data service. Two options are available:
   a. Run the sample Java client to populate the grid with data using the EntityManager API and query the data in the grid using a web browser and the eXtreme Scale REST data service. See "Using a Java client with REST data services" on page 218.
   b. Run the sample WCF Data Services C# client. See "Visual Studio 2008 WCF client with REST data service" on page 220.

Directory conventions

The following directory conventions are used throughout the documentation to must reference special directories such as *wxs_install_root* and *wxs_home*. You access these directories during several different scenarios, including during installation and use of command-line tools.

wxs_install_root

The *wxs_install_root* directory is the root directory where WebSphere eXtreme Scale product files are installed. The *wxs_install_root* directory can be the directory in which the trial zip file is extracted or the directory in which the WebSphere eXtreme Scale product is installed.

- Example when extracting the trial:

  Example: `/opt/IBM/WebSphere/eXtremeScale`
- Example when WebSphere eXtreme Scale is installed to a stand-alone directory:

  Example: `/opt/IBM/eXtremeScale`
- Example when WebSphere eXtreme Scale is integrated with WebSphere Application Server:

  Example: `/opt/IBM/WebSphere/AppServer`

wxs_home

The *wxs_home* directory is the root directory of the WebSphere eXtreme Scale product libraries, samples and components. This is the same as the *wxs_install_root* directory when the trial is extracted. For stand-alone installations, the *wxs_home* directory is the `ObjectGrid` sub-directory within the *wxs_install_root* directory. For installations that are integrated with WebSphere Application Server, this directory is the `optionalLibraries/ObjectGrid` directory within the *wxs_install_root* directory.

- Example when extracting the trial:

  Example: `/opt/IBM/WebSphere/eXtremeScale`
- Example when WebSphere eXtreme Scale is installed to a stand-alone directory:

  Example: `/opt/IBM/eXtremeScale/ObjectGrid`
- Example when WebSphere eXtreme Scale is integrated with WebSphere Application Server:

  Example: `/opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid`

was_root

The *was_root* directory is the root directory of a WebSphere Application Server installation:

Example: `/opt/IBM/WebSphere/AppServer`

restservice_home

The *restservice_home* directory is the directory in which the WebSphere eXtreme Scale REST data service libraries and samples are located. This directory is named `restservice` and is a sub-directory under the *wxs_home* directory.

- Example for stand-alone deployments:

  Example: `/opt/IBM/WebSphere/eXtremeScale/ObjectGrid/restservice`
- Example for WebSphere Application Server integrated deployments:

  Example: `/opt/IBM/WebSphere/AppServer/optionalLibraries/ObjectGrid/restservice`

tomcat_root

The *tomcat_root* is the root directory of the Apache Tomcat installation.

Example: `/opt/tomcat5.5`

wasce_root

The *wasce_root* is the root directory of the WebSphere Application Server Community Edition installation.

Example:`/opt/IBM/WebSphere/AppServerCE`

java_home

The *java_home* is the root directory of a Java Runtime Environment (JRE) installation.

Example:`/opt/IBM/WebSphere/eXtremeScale/java`

samples_home

The *samples_home* is the directory in which you extract the sample files that are used for tutorials.

Example:`/wxs-samples/`

Enabling the REST data service

The REST data service can represent WebSphere eXtreme Scale entity metadata to represent each entity as an EntitySet.

Starting a sample eXtreme Scale data grid

In general , before starting the REST data service, start the eXtreme Scale data grid. The following steps will start a single eXtreme Scale catalog service process and two container server processes.

WebSphere eXtreme Scale can be installed using three different methods:

- Trial install
- Stand-alone deployment
- WebSphere Application Server integrated deployment

Scalable data model in eXtreme Scale

The Microsoft Northwind sample uses the Order Detail table to establish a many-to-many association between Orders and Products.

Object to relational mapping specifications (ORMs) such as the ADO.NET Entity Framework and Java Persistence API (JPA) can map the tables and relationships using entities. However, this architecture does not scale. Everything must be located on the same machine, or an expensive cluster of machines to perform well.

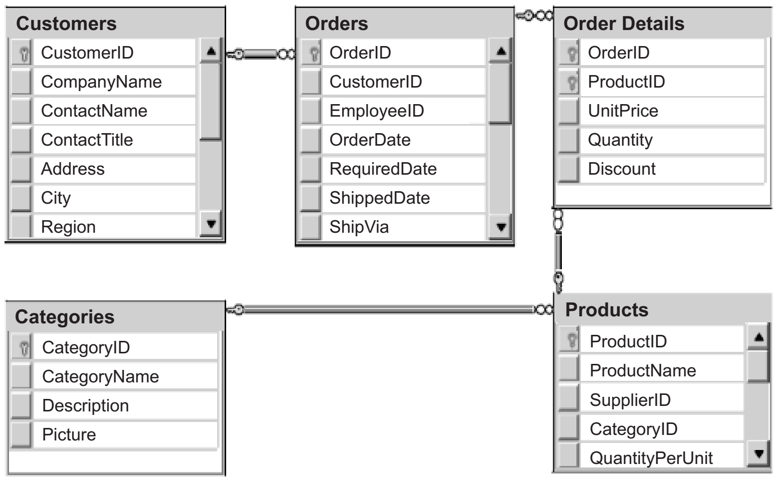


Figure 44. Microsoft SQL Server Northwind sample schema diagram

To create a scalable version of the sample, the entities must be modeled so each entity or group of related entities can be partitioned based off a single key. By creating partitions on a single key, requests can be spread out among multiple, independent servers. To achieve this configuration, the entities have been divided into two trees: the Customer and Order tree and the Product and Category tree. In this model, each tree can be partitioned independently and therefore can grow at different rates, increasing scalability.

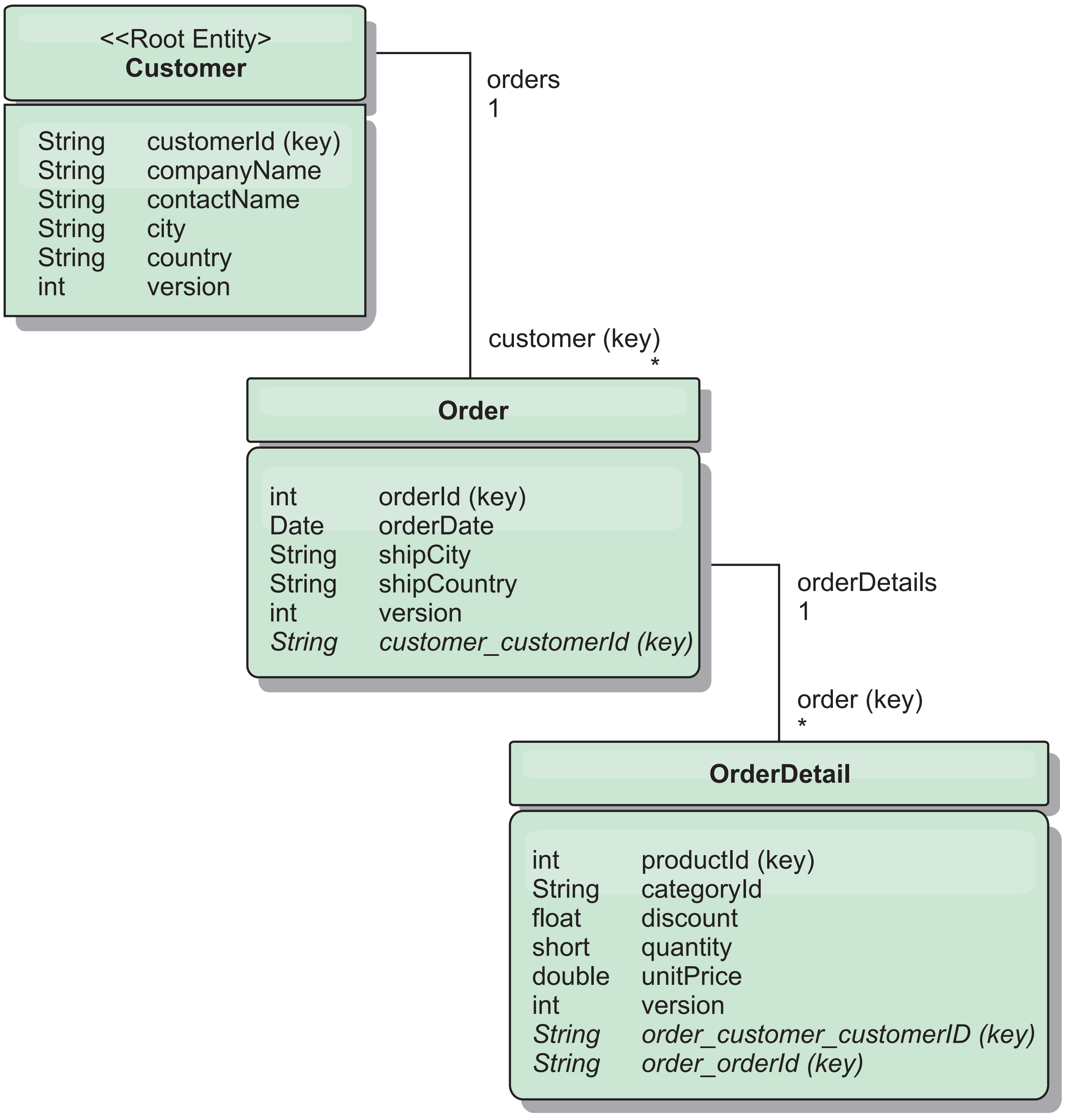


Figure 45. Customer and Order entity schema diagram

For example, both Order and Product have unique, separate integers as keys. In fact, the Order and Product tables are really independent of each other. For example, consider the effect of the size of a catalog, the number of products you sell, with the total number of orders. Intuitively, it might seem that having many products implies also having many orders, but this is not necessarily the case. If this were true, you could easily increase sales by just adding more products to your catalog. Orders and products have their own independent tables. You can further extend this concept so that orders and products each have their own separate, data grids. With independent data grids, you can control the number of partitions and servers, in addition to the size of each data grid separately so that your application can scale. If you double the size of your catalog, you must double the products data grid, but the order grid might be unchanged. The converse is true for an order surge, or expected order surge.

In the schema, a Customer has zero or more Orders, and an Order has line items (OrderDetail), each with one specific product. A Product is identified by ID (the Product key) in each OrderDetail. A single data grid stores Customers, Orders, and OrderDetails with Customer as the root entity of the data grid. You can retrieve Customers by ID, but you must get Orders starting with the Customer ID. So customer ID is added to Order as part of its key. Likewise, the customer ID and order ID are part of the OrderDetail ID.

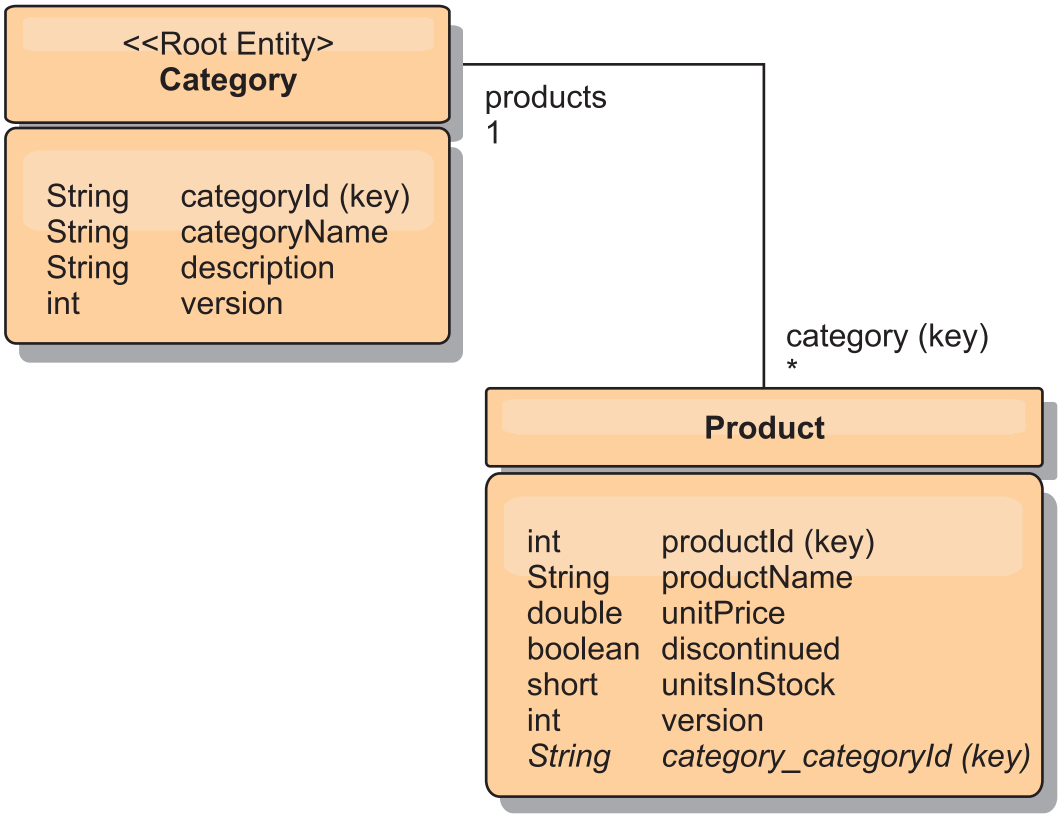


Figure 46. Category and Product entity schema diagram

In the Category and Product schema, the Category is the schema root. With this schema, customers can query products by category. See "Retrieving and updating data with REST" for additional details on key associations and their importance.

Retrieving and updating data with REST

The OData protocol requires that all entities can be addressed by their canonical form. This means that each entity must include the key of the partitioned, root entity, the schema root.

The following is an example of how to use the association from a root entity to address a child in :

```
/Customer('ACME')/order(100)
```

In WCF Data Services, the child entity must be directly addressable, meaning that the key in the schema root must be a part of the key of the child: `/Order(customer_customerId='ACME', orderId=100)`. This is achieved by creating an association to the root entity where the one-to-one or many-to-one association to the root entity is also identified as a key. When entities are included as part of the key, the attributes of the parent entity are exposed as key properties.

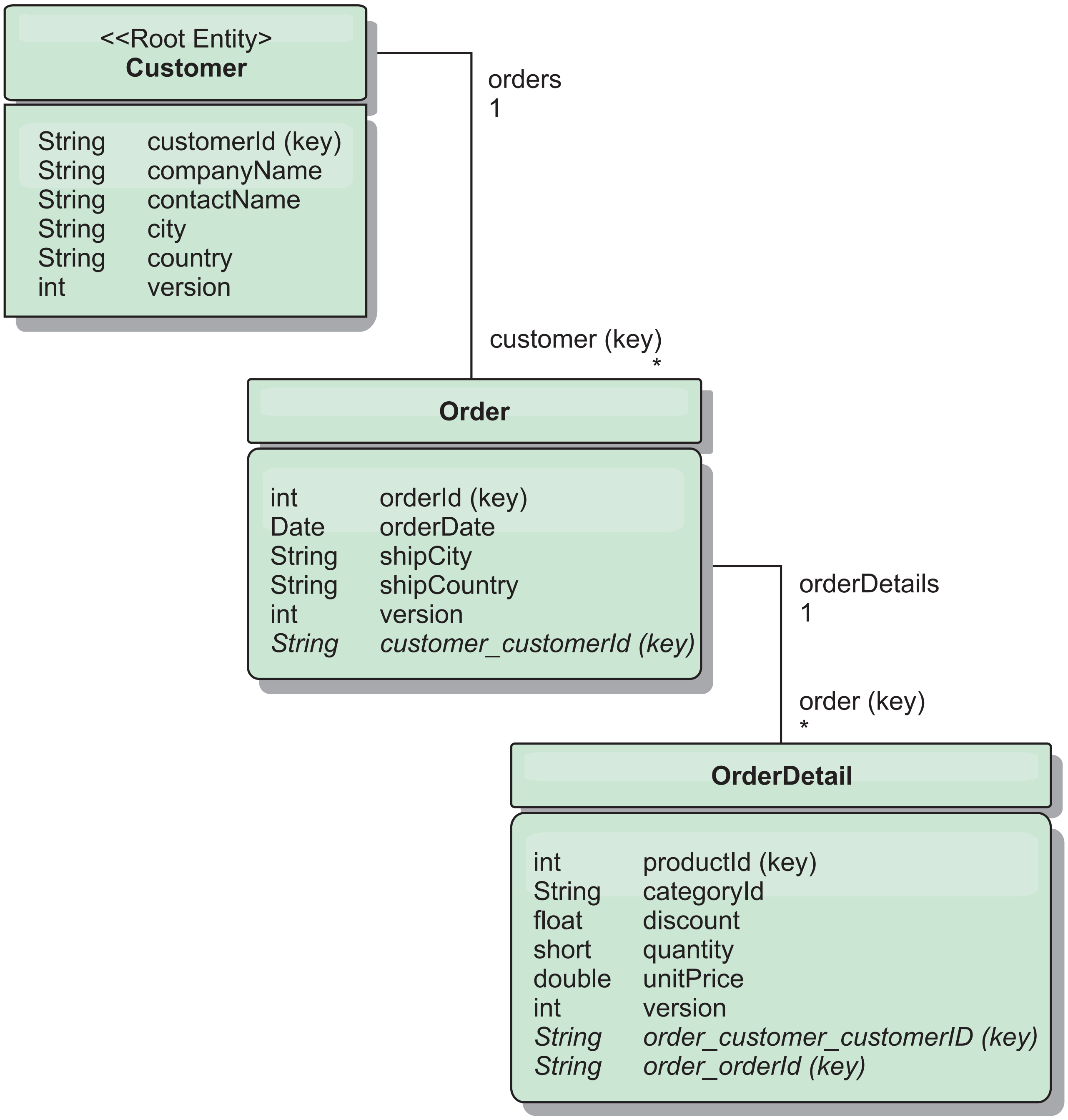


Figure 47. Customer and Order entity schema diagram

The Customer/Order entity schema diagram illustrates how each entity is partitioned using the Customer. The Order entity includes the Customer as part of its key and is therefore directly accessible. The REST data service exposes all key associations as individual properties: Order has customer_customerId and OrderDetail has order_customer_customerId and order_orderId.

Using the EntityManager API, you can find the Order using the Customer and order id:

```
transaction.begin();
// Look-up the Order using the Customer.  We only include the Id
// in the Customer class when building the OrderId key instance.
Order order = (Order) em.find(Order.class,
    new OrderId(100, new Customer('ACME')));
...
transaction.commit();
```

When using the REST data service, the Order can be retrieved with either of the URLs:

- `/Order(orderId=100, customer_customerId='ACME')`
- `/Customer('ACME')/orders?$filter=orderId eq 100`

The customer key is addressed using the attribute name of the Customer entity, an underscore character and the attribute name of the customer id: customer_customerId.

An entity can also include a non-root entity as part of its key if all of the ancestors to the non-root entity have key associations to the root. In this example, OrderDetail has a key-association to Order and Order has a key-association to the root Customer entity. Using the EntityManager API:

```
transaction.begin();
// Construct an OrderDetailId key instance.  It includes
// The Order and Customer with only the keys set.
Customer customerACME = new Customer("ACME");
Order order100 = new Order(100, customerACME);
OrderDetailId orderDetailKey =
    new OrderDetailId(order100, "COMP");
OrderDetail orderDetail = (OrderDetail)
    em.find(OrderDetail.class, orderDetailKey);
...
```

The REST data service allows addressing the OrderDetail directly:

```
/OrderDetail(productId=500, order_customer_customerId='ACME', order_orderId =100)
```

The association from the OrderDetail entity to the Product entity has been broken to allow partitioning the Orders and Product inventory independently. The OrderDetail entity stores the category and product id instead of a hard relationship. By decoupling the two entity schemas, only one partition is accessed at a time.

The Category and Product schema illustrated in the diagram shows that the root entity is Category and each Product has an association to a Category entity. The Category entity is included in the Product identity. The REST data service exposes a key property: category_categoryId which allows directly addressing the Product.

Because Category is the root entity, in a partitioned environment, the Category must be known in order to find the Product. Using the EntityManager API, the transaction must be pinned to the Category entity prior to finding the Product.

Using the EntityManager API:

```
transaction.begin();
// Create the Category root entity with only the key.  This
// allows us to construct a ProductId without needing to find
// The Category first.  The transaction is now pinned to the
// partition where Category "COMP" is stored.
```

```
Category cat = new Category("COMP");
Product product = (Product) em.find(Product.class,
    new ProductId(500, cat));
...
```

The REST data service allows addressing the Product directly:

```
/Product(productId=500, category_categoryId='COMP')
```

Starting a stand-alone data grid for REST data services

Follow these steps to start the WebSphere eXtreme Scale REST service sample data grid for a stand-alone eXtreme Scale deployment.

Before you begin

Install the WebSphere eXtreme Scale Trial or full product:

- Install the stand-alone version of the WebSphere eXtreme Scale 7.1 product and apply any subsequent fixes.
- Download and extract the WebSphere eXtreme Scale Version 7.1 trial, which includes the WebSphere eXtreme Scale REST data service.

About this task

Start the WebSphere eXtreme Scale sample data grid.

Procedure

1. Start the catalog service process. Open a command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*
2. `cd restservice_home/gettingstarted`
3. Start the catalog service process. To start the service *without* eXtreme Scale security, use the following commands.
   - Linux UNIX `./runcat.sh`
   - Windows `runcat.bat`

   To start the service witheXtreme Scale security, use the following commands.
   - Linux UNIX `./runcat_secure.sh`
   - Windows `runcat_secure.bat`
4. Start two container server processes. Open another command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*
5. `cd restservice_home/gettingstarted`
6. Start a container server process:

   To start the server without eXtreme Scale security, use the following commands:
   - Linux UNIX `./runcontainer.sh container0`
   - Windows `runcontainer.bat container0`

   To start the server witheXtreme Scale security, use the following commands.
   - Linux UNIX `./runcontainer_secure.sh container0`

- Windows `runcontainer_secure.bat container0`
7. Open another command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*
8. `cd restservice_home/gettingstarted`
9. Start a second container server process.

   To start the server without eXtreme Scale security, use the following commands.
   - Linux UNIX `./runcontainer.sh container1`
   - Windows `runcontainer.bat container1`

   To start the server with eXtreme Scale security, use the following commands.
   - Linux UNIX `./runcontainer_secure.sh container1`
   - Windows `runcontainer_secure.bat container1`

Results

Wait until the eXtreme Scale containers are ready before proceeding with the next steps. The container servers are ready when the following message is displayed in the terminal window:

```
CWOBJ1001I: ObjectGrid Server container_name is ready to process requests.
```

Where *container_name* is the name of the container that was started.

Starting a data grid for REST data services in WebSphere Application Server

Follow these steps to start a stand-alone WebSphere eXtreme Scale REST service sample data grid for a WebSphere eXtreme Scale deployment that is integrated with WebSphere Application Server. Although WebSphere eXtreme Scale is integrated with WebSphere Application Server, these steps start a stand-alone WebSphere eXtreme Scale catalog service process and container.

Before you begin

Install the WebSphere eXtreme Scale Version 7.1 product into a WebSphere Application Server Version 7.0.0.5 or later installation directory with security disabled. Augment at least one Application Server profile.

About this task

Start the WebSphere eXtreme Scale sample data grid.

Procedure

1. Start the catalog service process. Open a command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*

   `cd restservice_home/gettingstarted`
2. Start the catalog service process.

To start the server without eXtreme Scale security, use the following commands.

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

To start the server witheXtreme Scale security, use the following commands.

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

3. Start two container server processes. Open another command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*
4. Start a container server process.

   To start the server without eXtreme Scale security, use the following commands.
   a. Open a command-line window.
   b. `cd restservice_home/gettingstarted`
   c. To start the server *without* eXtreme Scale security, use the following commands.
      - Linux UNIX `./runcontainer.sh container0`
      - Windows `runcontainer.bat container0`
   d. To start the server with eXtreme Scale security, use the following commands.
      - Linux UNIX `./runcontainer_secure.sh container0`
      - Windows `runcontainer_secure.bat container0`
5. Start a second container server process.
   a. Open a command-line window.
   b. `cd restservice_home/gettingstarted`
   c. To start the server *without* eXtreme Scale security, use the following commands.
      - Linux UNIX `./runcontainer.sh container1`
      - Windows `runcontainer.bat container1`
   d. To start the server *with* eXtreme Scale security, use the following commands.
      - Linux UNIX `./runcontainer_secure.sh container1`
      - Windows `runcontainer_secure.bat container1`

Results

Wait until the container servers are ready before proceeding with the next steps. The container servers are ready when the following message is displayed:

```
CWOBJ1001I: ObjectGrid Server container_name is ready to process requests.
```

Where *container_name* is the name of the container that was started in the previous step.

Configuring application servers for the REST data service

`startOgServer`

Starting REST data services in WebSphere Application Server Version 7.0

This topic describes how to configure and start the eXtreme Scale REST data service usingWebSphere Application Server Version 7.0.

Before you begin

Verify that the sample eXtreme Scale data grid is started. See "Enabling the REST data service" on page 202 for details on how to start the data grid.

Procedure

1. Download and install WebSphere Application Server Version 7.0 for Developers.

   Restriction: Do not enable security.
2. Download and install WebSphere Application Server Version 7.0 Fix Pack 5 or later.
3. Add the WebSphere eXtreme Scale client runtime JAR, the `wsogclient.jar` file, and the REST data service configuration JAR or directory to the application server classpath:
   a. Open the WebSphere Application Server administrative console.
   b. Navigate **Environment** > **Shared libraries**
   c. Click **New**
   d. Add the following entries into the fields:
      1) Name: extremescale_client_v71
      2) Classpath: `wxs_home/lib/wsogclient.jar`
   e. Click **OK**
   f. Click **New**
   g. Add the following entries into the appropriate fields:
      1) Name: `extremescale_gettingstarted_config`
      2) Classpath:
         - `restservice_home/gettingstarted/restclient/bin`
         - `restservice_home/gettingstarted/common/bin`

      Remember: Add each path on a separate line.
   h. Click **OK**
   i. Save the changes to the master configuration
4. Install the REST data service EAR file, `wxsrestservice.ear`, to theWebSphere Application Server using the administrative console:
   a. Open the WebSphere Application Server administrative console
   b. Navigate to **Applications** > **New Application**
   c. Browse to the `restservice_home/lib/wxsrestservice.ear`, select the file and click **Next**
   d. Choose the detailed installation options, and click **Next**
   e. On the application security warnings screen, click **Continue**
   f. Choose the default installation options, and click **Next**
   g. Choose a server to map the application to, and click **Next**
   h. On the JSP reloading page, use the defaults, and click **Next**

i. On the shared libraries page, map the `wxsrestservice.war` module to the following defined shared libraries:
   - extremescale_client_v71
   - extremescale_ gettingstarted _config

j. On the map shared library relationship page, use the defaults, and click **Next**

k. On the map virtual hosts page, use the defaults, and click **Next**

l. On the map context roots page, set the context root to: `wxsrestservice`.

m. On the Summary screen, click **Finish** to complete the installation

n. Save the changes to the master configuration

5. Start the application server and the wxsrestservice eXtreme Scale REST data service application. After the application is started review the `SystemOut.log` file for the application server and verify that the following message displays: `CWOBJ4000I: The WebSphere eXtreme Scale REST data service has been started.`
6. Verify that the REST data service is working
   a. Open a browser and navigate to: `http://localhost:9080/wxsrestservice/restservice/NorthwindGrid`. The service document for the NorthwindGrid is displayed.
   b. Navigate to: `http://localhost:9080/wxsrestservice/restservice/NorthwindGrid/$metadata`. The Entity Model Data Extensions (EDMX) document is displayed.
7. To stop the data grid processes, use `CTRL+C` in the respective command window.

Starting REST data services with WebSphere eXtreme Scale integrated in WebSphere Application Server 7.0

This topic describes how to configure and start the eXtreme Scale REST data service using WebSphere Application Server version 7.0 that has been integrated and augmented with WebSphere eXtreme Scale.

Before you begin

Verify that the sample stand-alone eXtreme Scale data grid is started. See "Enabling the REST data service" on page 202 for details on how to start the data grid.

About this task

To get started with the WebSphere eXtreme Scale REST data service using WebSphere Application Server, follow these steps:

Procedure

1. Add the WebSphere eXtreme Scale REST data service sample configuration JAR to the classpath:
   a. Open the WebSphere Administration Console
   b. Navigate to Environment -> Shared libraries
   c. Click New
   d. Add the following entries into the appropriate fields:
      1) Name: extremescale_gettingstarted_config
      2) Classpath
         - `restservice_home/gettingstarted/restclient/bin`
         - `restservice_home/gettingstarted/common/bin`

Remember: Each path must appear on a different line.

e. Click **OK**

f. Save the changes to the master configuration

2. Install the REST data service EAR file, wxsrestservice.ear, to the WebSphere Application Server using the WebSphere administration console:

   a. Open the WebSphere administration console

   b. Navigate to Applications -> New Application

   c. Browse to `restservice_home/lib/wxsrestservice.ear` file on the file system. Select the file and click **Next**.

   d. Choose the detailed installation options, and click **Next**.

   e. On the application security warnings screen, click **Continue**.

   f. Choose the default installation options, and click **Next**.

   g. Choose a server to map the wxsrestservice.war module to, and click **Next**.

   h. On the JSP reloading page, use the defaults, and click **Next**.

   i. On the shared libraries page, map the "wxsrestservice.war" module to the following shared libraries that were defined during step 1: `extremescale_ gettingstarted _config`

   j. On the map shared library relationship page, use the defaults, and click **Next**.

   k. On the map virtual hosts page, use the defaults, and click **Next**.

   l. On the map context roots page, set the context root to: wxsrestservice.

   m. On the Summary screen, click **Finish** to complete the installation.

   n. Save the changes to the master configuration.

3. If the eXtreme Scale data grid was started with eXtreme Scale security enabled, set the following property in the `restservice_home/gettingstarted/restclient/bin/wxsRestService.properties` file.

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
```

4. Start the application server and the "wxsrestservice " eXtreme Scale REST data service application.

   After the application is started review the SystemOut.log for the application server and verify that the following message appears: `CWOBJ4000I: The WebSphere eXtreme Scale REST data service has been started.`

5. Verify that the REST data service is working:

   a. Open a browser and navigate to:

      http://localhost:9080/wxsrestservice/restservice/NorthwindGrid

      The service document for the NorthwindGrid is displayed.

   b. Navigate to:

      http://localhost:9080/wxsrestservice/restservice/NorthwindGrid/$metadata

      The Entity Model Data Extensions (EDMX) document is displayed

6. To stop the data grid processes, use CTRL+C in the respective command window to stop the process.

Starting the REST data service in WebSphere Application Server Community Edition

This topic describes how to configure and start the eXtreme Scale REST data service using WebSphere Application Server Community Edition.

Before you begin

Verify that the sample data grid is started. See "Enabling the REST data service" on page 202 for details on how to start the grid.

Procedure

1. Download and install WebSphere Application Server Community Edition Version 2.1.1.3 or later to `wasce_root`, such as: `/opt/IBM/wasce`
2. Start the WebSphere Application Server Community Edition server by running the following command:
   - Linux UNIX `wasce_root/bin/startup.sh`
   - Windows `wasce_root/bin/startup.bat`
3. If the eXtreme Scale grid was started with eXtreme Scale security enabled, set the following properties in the `restservice_home/gettingstarted/restclient/bin/wxsRestService.properties` file.

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
loginType=none
```

4. Install the eXtreme Scale REST data service and the provided sample into the WebSphere Application Server Community Edition server:
   a. Add the ObjectGrid client runtime JAR to the WebSphere Application Server Community Edition repository:
      1) Open the WebSphere Application Server Community Edition administration console and log in.

         Tip: The default URL is: `http://localhost:8080/console`. The default user ID is `system` and password is `manager`.
      2) Click the **Repository**, in the Services folder.
      3) In the **Add Archive to Repository** section, fill in the following into the input text boxes:

Table 15. Archive to repository

Text box	Value
File	wxs_home/lib/ogclient.jar
Group	com.ibm.websphere.xs
Artifact	ogclient
Version	7.0
Type	jar

      4) Click the Install button.

         Tip: See the following tech note for details on different methods of configuration class and library dependencies: Specifying external dependencies to applications running on WebSphere Application Server Community Edition.
   b. Deploy the REST data service module, which is the `wxsrestservice.war` file, to the WebSphere Application Server Community Edition server.
      1) Edit the sample `restservice_home/gettingstarted/wasce/geronimo-web.xml` deployment XML file to include path dependencies to the getting started sample classpath directories:

         Change the classesDirs paths for the two getting started client GBeans:

- The "classesDirs" path for the GettingStarted_Client_SharedLib GBean should be set to: `restservice_home/gettingstarted/restclient/bin`
- The "classesDirs" path for the GettingStarted_Common_SharedLib GBean should be set to: `restservice_home/gettingstarted/common/bin`

2) Open the WebSphere Application Server Community Edition administrative console and log in.

   Tip: The default URL is: `http://localhost:8080/console`. The default user ID is `system` and password is `manager`.
3) Click **Deploy New**.
4) On the **Install New Applications** page, enter the following values into the text boxes:

Table 16. Installation values

Text box	Value
Archive	`restservice_home/lib/wxsrestservice.war`
Plan	`restservice_home/gettingstarted/wasce/geronimo-web.xml`

5) Click on the Install button.

   The console page should indicate that the application was successfully installed and started.
6) Examine the WebSphere Application Server Community Edition system output log or console to verify that the REST data service has started successfully by verify that the following message is present:

```
CWOBJ4000I: The WebSphere eXtreme Scale REST data service has
been started.
```

5. Verify that the REST data service is working:
   a. Open the following link in a browser window: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid`. The service document for the NorthwindGrid grid is displayed.
   b. Open the following link in a browser window: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/$metadata`. The Entity Model Data Extensions (EDMX) document is displayed.
6. To stop the grid processes, use `CTRL+C` in the respective command window to stop the process.
7. To stop WebSphere Application Server Community Edition, use the following command:
   - UNIX Linux `wasce_root/bin/shutdown.sh`
   - Windows `wasce_root\bin\shutdown.bat`

   Tip: The default user ID is `system` and password is `manager`. If you are using a custom port, use the `-port` option.

Starting REST data services in Apache Tomcat

This topic describes how to configure and start the eXtreme Scale REST data service using Apache Tomcat, version 5.5 or later.

Before you begin

Verify that the sample eXtreme Scale data grid is started. See "Enabling the REST data service" on page 202 for details on how to start the data grid.

Procedure

1. Download and install Apache Tomcat Version 5.5 or later to tomcat_root. For example: `/opt/tomcat`
2. Install the eXtreme Scale REST data service and the provided sample into the Tomcat server as follows:
   a. If you are using a Sun JRE or JDK, you must install the IBM ORB into Tomcat:
      - For Tomcat version 5.5

        Copy all of the JAR files from:

        `wxs_home/lib/endorsed`

        to

        `tomcat_root/common/endorsed`
      - For Tomcat version 6.0
        1) Create an "endorsed" directory
           - UNIX Linux `mkdir tomcat_root/endorsed`
           - Windows `md tomcat_root/endorsed`
        2) Copy all of the JAR files from:

           `wxs_home/lib/endorsed`

           to

           `tomcat_root/endorsed`

   b. Deploy the REST data service module: wxsrestservice.war to the Tomcat server.

      Copy the wxsrestservice.war file from:

      `restservice_home/lib`

      to:

      `tomcat_root/webapps`

   c. Add the ObjectGrid client runtime JAR and the application JAR to the shared classpath in Tomcat:
      1) Edit the `tomcat_root/conf/catalina.properties` file
      2) Append the following path names to the end of the shared.loader property in the form of a comma-delimited list:
         - wxs_home/lib/ogclient.jar
         - restservice_home/gettingstarted/restclient/bin
         - restservice_home/gettingstarted/common/bin

         Important: The path separator must be a **forward** slash.
3. If the eXtreme Scale data grid was started with eXtreme Scale security enabled, set the following properties in the `restservice_home/gettingstarted/restclient/bin/wxsRestService.properties` file.

```
ogClientPropertyFile=restservice_home/gettingstarted/security/security.ogclient.properties
loginType=none
```

4. Start the Tomcat server with the REST data service:
   - If using Tomcat 5.5 on UNIX or Windows, or Tomcat 6.0 on UNIX:

a. `cd tomcat_root/bin`

b. Start the server:

   – UNIX Linux `./catalina.sh run`

   – Windows `catalina.bat run`

c. The console then displays the Apache Tomcat logs. When the REST data service has started successfully, the following message is displayed in the administration console:

   ```
   CWOBJ4000I: The WebSphere eXtreme Scale REST data service has been
   started.
   ```

- If using Tomcat 6.0 on Windows:

  a. `cd tomcat_root/bin`

  b. Start the Apache Tomcat 6 configuration tool with the following command: `tomcat6w.exe`

  c. Click on the Start button on the Apache Tomcat 6 properties window to start the Tomcat server.

  d. Review the following logs to verify that the Tomcat server has started successfully:

     – `tomcat_root/bin/catalina.log`

       Displays the status of the Tomcat server engine

     – `tomcat_root/bin/stdout.log`

       Displays the system output log.

  e. When the REST data service has started successfully, the following message is displayed in the system output log: `CWOBJ4000I: The WebSphere eXtreme Scale REST data service has been started.`

5. Verify that the REST data service is working:

   a. Open a browser and navigate to:

      http://localhost:8080/wxsrestservice/restservice/NorthwindGrid

      The service document for the NorthwindGrid is displayed.

   b. Navigate to:

      http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
      $metadata

      The Entity Model Data Extensions (EDMX) document is displayed.

6. To stop the data grid processes, use CTRL+C in the respective command window.

7. To stop Tomcat, use CTRL +C in the window in which you started it.

Using a browser with REST data services

The eXtreme Scale REST data service creates ATOM feeds by default when using a web browser. The ATOM feed format may not be compatible with older browsers or may be interpreted such that the data cannot be viewed as XML. The following topics provide details on how to configure Internet Explorer Version 8 and Firefox Version 3 to display the ATOM feeds and XML within the browser.

About this task

The eXtreme Scale REST data service creates ATOM feeds by default when using a web browser. The ATOM feed format may not be compatible with older browsers or may be interpreted such that the data cannot be viewed as XML. For older browsers, you will be prompted to save the files to disk. Once the files are downloaded, use your favorite XML reader to look at the files. The generated XML is not formatted to be displayed, so everything will be printed on one line. Most XML reading programs, such as Eclipse, support reformatting the XML into a readable format.

For modern browsers, such as Microsoft Internet Explorer Version 8 and Firefox Version 3, the ATOM XML files can be displayed natively in the browser. The following topics provide details on how to configure Internet Explorer Version 8 and Firefox Version 3 to display the ATOM feeds and XML within the browser.

Procedure

Configuring Internet Explorer Version 8

- To enable Internet Explorer to read the ATOM feeds that the REST data service generates use the following steps.:
  1. Click **Tools** > **Internet Options**
  2. Select the **Content** tab
  3. Click the **Settings** button in the **Feeds and Web Slices** section
  4. Uncheck the box: "Turn on feed reading view"
  5. Click **OK** to return to the browser.
  6. Restart Internet Explorer.

Configuring Firefox Version 3

- Firefox does not automatically display pages with content type: application/atom+xml. The first time a page is displayed, Firefox prompts you to save the file. To display the page, open the file itself with Firefox as follows:
  1. From the application chooser dialog box, select the "Open with" radio button and click the **Browse** button.
  2. Navigate to your Firefox installation directory. For example: `C:\Program Files\Mozilla Firefox`
  3. Select `firefox.exe` and hit the **OK** button.
  4. Check the “Do this automatically for files like this...” check box.
  5. Click the **OK** button.
  6. Next, Firefox displays the ATOM XML page in a new browser window or tab
- Firefox automatically renders ATOM feeds in readable format. However, the feeds that the REST data service creates include XML. Firefox cannot display the XML unless you disable the feed renderer. Unlike Internet Explorer, in Firefox, the ATOM feed rendering plug-in must be explicitly edited. To configure Firefox to read ATOM feeds as XML files, follow these steps:
  1. Open the following file in a text editor: `<firefoxInstallRoot>\components\FeedConverter.js`. In the path, `<firefoxInstallRoot>` is the root directory where Firefox is installed.

     For Windows operating systems, the default directory is: `C:\Program Files\Mozilla Firefox`.
  2. Search for the snippet that looks as follows:
     ```
     // show the feed page if it wasn't sniffed and we have a document,
     // or we have a document, title, and link or id
     if (result.doc && (!this._sniffed ||
         (result.doc.title && (result.doc.link || result.doc.id)))) {
     ```
  3. Comment out the two lines that begin with `if` and `result` by placing `//` (two forward slashes) in front of them.
  4. Append the following statement to the snippet: `if(0) {`.

5. The resulting text should look as follows:

   ```
   // show the feed page if it wasn't sniffed and we have a document,
   // or we have a document, title, and link or id
   //if (result.doc && (!this._sniffed ||
   //    (result.doc.title && (result.doc.link || result.doc.id)))) {
   if(0) {
   ```
6. Save the file.
7. Restart Firefox
8. Now Firefox can automatically display all feeds in the browser.

- Test your setup by trying some URLs.

Example

This section describes some example URLs that can be used to view the data that was added by the getting started sample provided with the eXtreme Scale REST data service. Before using the following URLs, add the default data set to the eXtreme Scale sample grid using either the sample Java client or the sample Visual Studio WCF Data Services client.

The following examples assume the port is 8080 which can vary. See section for details on how to configure the REST data service on different application servers.

- View a single customer with the id of "ACME":
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')
  ```
- View all of the orders for customer "ACME":
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')/orders
  ```
- View the customer "ACME" and the orders:
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')?$expand=orders
  ```
- View order 1000 for customer "ACME":
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=1000,customer_customerId='ACME')
  ```
- View order 1000 for customer "ACME" and its associated Customer:
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Order(orderId=1000,customer_customerId='ACME')?$expand=customer
  ```
- View order 1000 for customer "ACME" and its associated Customer and OrderDetails:
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Order(orderId=1000,customer_customerId='ACME')?$expand=customer,orderDetails
  ```
- View all orders for customer "ACME" for the month of October, 2009 (GMT):
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Customer(customerId='ACME')/orders?$filter=orderDate
  ge datetime'2009-10-01T00:00:00'
  and orderDate lt datetime'2009-11-01T00:00:00'
  ```
- View all the first 3 orders and orderDetails for customer "ACME" for the month of October, 2009 (GMT):
  ```
  http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Customer(customerId='ACME')/orders?$filter=orderDate
  ge datetime'2009-10-01T00:00:00'
  and orderDate lt datetime'2009-11-01T00:00:00'
  &$orderby=orderDate&$top=3&$expand=orderDetails
  ```

Using a Java client with REST data services

The Java client application uses the eXtreme Scale EntityManager API to insert data into the grid.

About this task

The previous sections described how to create an eXtreme Scale data grid and configure and start the eXtreme Scale REST data service. The Java client application uses the eXtreme Scale EntityManager API to insert data into the grid. It does not demonstrate how to use the REST interfaces. The purpose of this client is to demonstrate how the EntityManager API is used to interact with the eXtreme Scale data grid, and allow modifying data in the grid. To view data in the grid using the REST data service, use a web browser or use the Visual Studio 2008 client application.

Procedure

To quickly add content to the eXtreme Scale data grid, run the following command:

1. Open a command-line or terminal window and set the JAVA_HOME environment variable:
   - Linux UNIX `export JAVA_HOME=`*`java_home`*
   - Windows `set JAVA_HOME=`*`java_home`*
2. `cd restservice_home/gettingstarted`
3. Insert some data into the grid. The data that is inserted will be retrieved later using a Web browser and the REST data service.

   If the data grid was started *without*eXtreme Scale security, use the following commands.
   - UNIX Linux `./runclient.sh load default`
   - Windows `runclient.bat load default`

   If the data grid was started *with*eXtreme Scale security, use the following commands.
   - UNIX Linux `./runclient_secure.sh load default`
   - Windows `runclient_secure.bat load default`

   For a Java client, use the following command syntax:
   - UNIX Linux `runclient.sh` *`command`*
   - Windows `runclient.bat` *`command`*

   The following commands are available:
   - `load default`

     Loads a predefined set of Customer, Category and Product entities into the data grid and creates a random set of Orders for each customer.
   - `load category` *`categoryId categoryName firstProductId num_products`*

     Creates a product Category and a fixed number of Product entities in the data grid. The firstProductId parameter identifies the id number of the the first product and each subsequent product is assigned the next id until the specified number of products is created.
   - `load customer` *`companyCode contactNamecompanyName numOrders firstOrderIdshipCity maxItems discountPct`*

     Loads a new Customer into the data grid and creates a fixed set of Order entities for any random product currently loaded in the grid. The number of Orders is determined by setting the <numOrders> parameter. Each Order will have a random number of OrderDetail entities up to <maxItems>

- `display customer` *`companyCode`*

  Display a Customer entity and the associated Order and OrderDetail entities.
- `display category` *`categoryId`*

  Display a product Category entity and the associated Product entities.

Results

- `runclient.bat load default`
- `runclient.bat load customer IBM "John Doe" "IBM Corporation" 5 5000 Rochester 5 0.05`
- `runclient.bat load category 5 "Household Items" 100 5`
- `runclient.bat display customer IBM`
- `runclient.bat display category 5`

Running and building the sample data grid and Java client with Eclipse

The REST data service getting started sample can be updated and enhanced using Eclipse. For details on how to setup your Eclipse environment see the text document: `restservice_home/gettingstarted/ECLIPSE_README.txt`.

After the WXSRestGettingStarted project is imported into Eclipse and is building successfully, the sample will automatically re-compile and the script files used to start the container server and client will automatically pick up the class files and XML files. The REST data service will also automatically detect any changes since the Web server is configured to read the Eclipse build directories automatically.

Important: When changing source or configuration files, both the eXtreme Scale container server and the REST data service application must be restarted. The eXtreme Scale container server must be started before the REST data service Web application.

Visual Studio 2008 WCF client with REST data service

The eXtreme Scale REST data service getting started sample includes a WCF Data Services client that can interact with the eXtreme Scale REST data service. The sample is written as a command-line application in C#.

Software requirements

The WCF Data Services C# sample client requires the following:

- Operating system
  - Microsoft Windows XP
  - Microsoft Windows Server 2003
  - Microsoft Windows Server 2008
  - Microsoft Windows Vista
- Microsoft Visual Studio 2008 with Service Pack 1

  Tip: See the previous link for additional hardware and software requirements.
- Microsoft .NET Framework 3.5 Service Pack 1
- Microsoft Support: An update for the .NET Framework 3.5 Service Pack 1 is available

Building and running the getting started client

The WCF Data Services sample client includes a Visual Studio 2008 project and solution and the source code for running the sample. The sample must be loaded into Visual Studio 2008 and compiled into a Windows runnable program before it can be run. To build and run the sample, see the text document: `restservice_home/gettingstarted/VS2008_README.txt`.

WCF Data Services C# client command syntax

Windows `WXSRESTGettingStarted.exe <service URL> <command>`

The <service URL> is the URL of the eXtreme Scale REST data service configured in section .

The following commands are available:

- `load default`

  Loads a predefined set of Customer, Category and Product entities into the data grid and creates a random set of Orders for each customer.
- `load category <categoryId> <categoryName> <firstProductId> <numProducts>`

  Creates a product Category and a fixed number of Product entities in the data grid. The firstProductId parameter identifies the id number of the the first product and each subsequent product is assigned the next id until the specified number of products is created.
- `load customer <companyCode> <contactName> <companyName> <numOrders> <firstOrderId> <shipCity> <maxItems> <discountPct>`

  Loads a new Customer into the data grid and creates a fixed set of Order entities for any random product currently loaded in the data grid. The number of Orders is determined by setting the <numOrders> parameter. Each Order will have a random number of OrderDetail entities up to <maxItems>
- `display customer <companyCode>`

  Display a Customer entity and the associated Order and OrderDetail entities.
- `display category <categoryId>`

  Display a product Category entity and the associated Product entities.
- `unload`

  Remove all entities that were loaded using the "default load" command.

The following examples illustrate various commands.

- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load default`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load customer`
- `IBM "John Doe" "IBM Corporation" 5 5000 Rochester 5 0.05`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid load category 5 "Household Items" 100 5`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid display customer IBM`
- `WXSRestGettingStarted.exe http://localhost:8080/wxsrestservice/restservice/NorthwindGrid display category 5`

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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