IBM WebSphere eXtreme Scale Version 7.0

Product Overview

March 11, 2011

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

Contents

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

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

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

Who should use this book

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

How this book is structured

The book contains information about the following major topics:

- **Chapter 1** includes an overview of WebSphere eXtreme Scale
- **Chapter 2** includes information about caching concepts in the product.
- **Chapter 3** includes information about cache integration.
- **Chapter 4** includes information about scalability.
- **Chapter 5** includes information about availability.
- **Chapter 6** includes information about security.
- **Chapter 7** includes information about transaction processing.
- **Chapter 8** includes tutorials for basic product concepts.
- **Chapter 9** includes the product glossary.

Getting updates to this book

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

How to send your comments

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

Chapter 1. WebSphere eXtreme Scale 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. WebSphere eXtreme Scale performs massive volumes of transaction processing with high efficiency and linear scalability, and provides qualities of service such as transactional integrity, high availability, and predictable response times.

The elastic scalability is possible through the use of distributed object caching. Elastic means the grid monitors and manages itself, allows scale-out and scale-in, and is self-healing by automatically recovering from failures. Scale-out allows memory capacity to be added while the grid is running, without requiring a restart. Conversely, scale-in allows for immediate removal of memory capacity.

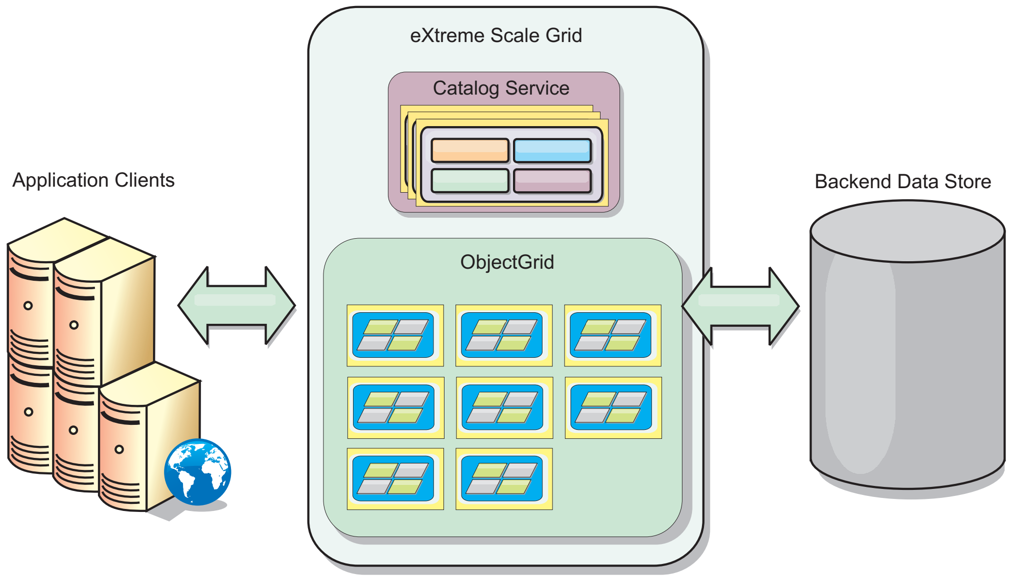
WebSphere eXtreme Scale can be used in different ways. It can be used as a very powerful cache or as a form of an in-memory database processing space to manage application state or as a platform for building powerful Extreme Transaction Processing (XTP) applications.

It is important to note, however, that eXtreme Scale cannot be considered an actual in-memory database, much of which is because the latter is too simple to handle some of the complexities that eXtreme Scale can manage. You would have some of the same advantages with both scenarios because they are each in memory, but if an in-memory database has a machine that fails, it is not immediately capable of repairing the issue. Such an event would be particularly disastrous if your entire environment is on that one machine.

To tackle the problem of this type of failure, eXtreme Scale splits the given dataset into partitions, which are equivalent to constrained tree schemas. Each partition exists as a primary copy, or shard, and also replica shards for backing up the data. An in-memory database cannot provide this kind of functionality because it is not structured and dynamic in this way, requiring you to do manually what eXtreme Scale does automatically. Furthermore, because an in-memory database is a database, it can allow SQL operations, and works as a great improvement in terms of processing speed compared to databases that are not in memory. WebSphere eXtreme Scale has its own query language rather than SQL support, but is significantly more elastic, allows for partitioning of data, and provides dependable failure recovery.

The write-behind cache feature allows WebSphere eXtreme Scale to serve as a front-end cache for a database to increase throughput 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 grid clients send and receive data from the grid, which 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, preventing wasteful copies of records that could potentially contain different versions of the data. A coherent cache holds more data as more servers are added to the grid, and scales linearly as the grid grows in size. The data can also be optionally replicated for additional fault tolerance.



WebSphere eXtreme Scale has 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, allowing more than one of these per physical machine. Thus, the in-memory data grid may be quite large. The 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 machines.

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

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

Basic paradigm

The fundamental grid paradigm is a key-value pair, where the grid stores values (Java objects), with an associated key (another Java object), by which the value is subsequently retrieved. In eXtreme Scale, a map consists of entries of such key-value pairs.

WebSphere eXtreme Scale offers a number of 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, objects with relationships can be stored. You can use a query language that is similar to 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, much like Java Enterprise Edition entities. Entity relationships may be automatically discovered from an entity descriptor XML file or annotations in the Java classes. Thus, an entity may be retrieved from the cache by primary key using the EntityManager find method. Entities may be persisted to the grid, as well as removed from it, all within a transaction boundary.

WebSphere eXtreme Scale provides extreme transaction processing (XTP) capabilities that ensure a smarter application infrastructure to support your most demanding business-critical applications. You can overcome traditional IT performance limitations to generate the levels of global scale, process efficiencies, and business intelligence needed for smarter outcomes and for sustainable competitive business advantage.

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

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.

Consider a distributed example where the key is a simple alphabetic name. The cache might be split into 4 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 (is stored in) a primary shard and a replica shard. Changes to the cache data are made to the primary shard, and replicated to the secondary shard. For a distributed cache, (or grid or ObjectGrid in eXtreme Scale vocabulary), you configure the number of eXtreme Scale servers that will contain the grid data, and eXtreme Scale distributes the data into shards over these server instances. For availability, replica shards are placed in separate machines 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 should they or their containing physical machines fail and subsequently 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, and, as before, a new replica shard is constructed.

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

For a discussion of the best practices that you can use when you are designing your WebSphere eXtreme Scale applications, read the following article on developerWorks®: Principles and best practices for building high performing and highly resilient WebSphere eXtreme Scale applications.

New and deprecated features in this release

WebSphere eXtreme Scale includes many new features in Version 7.0, including integration with the dynamic cache, byte array maps, and more.

What is new in WebSphere eXtreme Scale Version 7.0

Table 1. New features in WebSphere eXtreme Scale Version 7.0

Feature	Description
Dynamic cache integration	The dynamic cache provider allows applications that are using the WebSphere Application Server dynamic cache to leverage the advanced features and performance improvements of WebSphere eXtreme Scale. This feature results in a higher quality of service, linear scalability and high availability to bear on a broad variety of business applications with minimal invasive changes. See the information about dynamic cache in the *Product Overview*.
Byte array maps	With byte array maps, the application can store the value of a key-value pair in a byte array instead of object form, which reduces the memory footprint that a large graph of objects can consume. See the information about byte array maps in the *Programming Guide*.
WebSphere Real Time	With support for WebSphere Real Time, the industry-leading real-time Java offering, WebSphere eXtreme Scale enables XTP applications to have more consistent and predictable response times. See the information about Real Time Support in the *Administration Guide*.
Metric enablement	WebSphere eXtreme Scale includes implementations of metric access adapters to improve integration with IBM® Tivoli® Monitoring (ITM) and Hyperic HQ, enabling comprehensive insight into the operational behavior of business solutions. See the information on vendor monitoring tools for metric enablement in the *Administration Guide*..
Dynamic maps	Building multi-tenant applications has been greatly simplified. Map templates allow applications to create new maps on demand, avoiding the need to use application discriminators in the keys or creating extra maps that may never be used. See the information on dynamic maps in the *Programming Guide*..
Request timeout	WebSphere eXtreme Scale is enhanced to handle many of the common retry and exception logic tasks within the grid middleware. The request timeout for clients removes the burden from developers for boilerplate retry logic for most map interaction operations. Most retry-able conditions are now handled automatically, so you can focus on the business logic aspects of application development. See the information on request timeout in the *Administration Guide*..
Composite index	This feature can simplify index usage when querying multiple attributes and reduce the overhead of having multiple indexes defined. Query has also been optimized to take advantage of composite indexes. See the information on composite index in the *Programming Guide*..

Deprecated features

Table 2. Deprecated features

Deprecation	Recommended migration action
Partitioning facility (WPF): The partitioning facility is a set of programming APIs that allow Java EE applications to support asymmetric clustering.	The capabilities of WPF can be alternatively realized in WebSphere eXtreme Scale.
StreamQuery: A continuous query over in-flight data stored in ObjectGrid maps.	None
Static grid configuration: A static, cluster-based topology using the cluster deployment XML file.	Replaced with the improved, dynamic deployment topology for managing large data grids.
Deprecated system properties: System properties to specify the server and client properties files are deprecated.	You can still use these arguments, but change your system properties to the new values. See the information about the properties files in the *Administration Guide* for more information.

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.

Working with WebSphere eXtreme Scale

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.

Since it is not an in-memory database, you must consider the specific configuration requirements for eXtreme Scale. The first step to deploying an eXtreme Scale data grid is to start a core group and catalog service, which will act as coordinator for all other Java Virtual Machines participating in the 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 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 grid services. More servers increase both grid capacity and reliability.

A local grid is a simple, single-instance grid where all the data is in the one grid. To effectively use eXtreme Scale as an in-memory database processing space, you can configure and deploy a distributed 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 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 grid. A server may contain one or more 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 grid. The maximum capacity of a grid is the number of partitions times the usable memory size of a server, which may be a JVM.

A partition's data is stored in a shard. For availability, a grid may be configured with replicas, which may be synchronous or asynchronous. Changes to the grid data are made to the primary shard, and replicated to the replica shards. The total memory consumed/required by a grid is thus the size of the 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 containing the grid. These servers may be on the same and/or separate physical machines. For availability, replica shards are placed in separate machines from primary shards.

WebSphere eXtreme Scale monitors the status of its servers and moves shards among them should they and/or their containing physical machines fail and subsequently recover. For example, if the server containing a replica shard fails, eXtreme Scale will allocate a new replica shard, and replicate data from the primary to the new replica. Should a server containing a primary shard fail, the replica shard is promoted to be the primary shard, and, as before, a new replica shard is constructed. If you start an additional server for the data grid, the shards will be distributed over all servers so that the load on each is as balanced as possible. This is called scale-out. Similarly, for scale-in, you may stop one of the servers to reduce the resources consumed by a data grid, and again the shards will be balanced over the remaining servers, just as in a failure situation.

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.

Planning application deployment

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

Planning application deployment

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?

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 eXtreme Scale grid.

Integrating with other WebSphere Application Server products

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

Configuring the HTTP session manager to work with 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. See the *Administration Guide* for more details.

Configuring the WebSphere eXtreme Scale session manager to work with WebSphere Application Server

The HTTP session manager first shipped with WebSphere Extended Deployment DataGrid Version 6.1.0.0. Subsequent versions through Version 6.1.0.5 have not changed the utilization methods, since it meets the Java 2 Enterprise Edition (J2EE) specification for integration and session retrieval, but there have been performance and QoS enhancements with every release. To ensure you are getting the best quality of service, the recommendation is to apply the WebSphere eXtreme Scale version 6.1.0.5 Fix Pack.

See the *Administration Guide* for more details.

Chapter 2. Caching concepts

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 eXtreme Scale. This section describes various eXtreme Scale cache concepts and scenarios and discusses the available topologies for deploying an eXtreme Scale grid.

Architecture and topology

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.

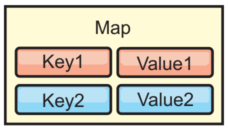


Figure 1. Map

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.

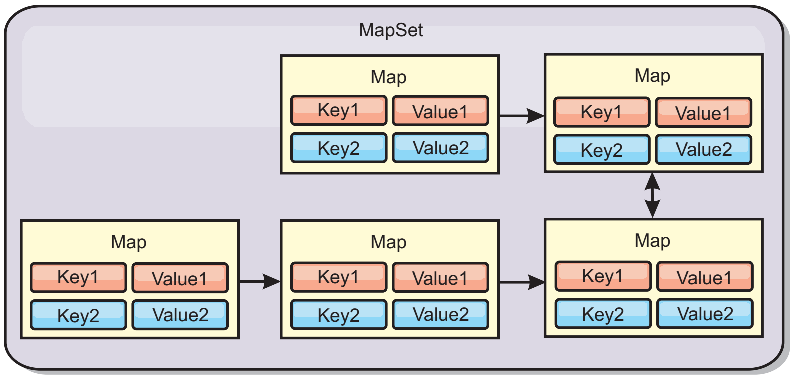


Figure 2. Map sets

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.

Containers, partitions, and shards

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

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

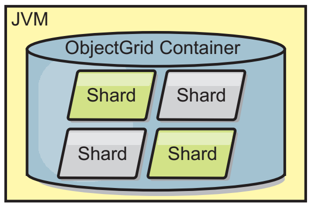


Figure 3. Container

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

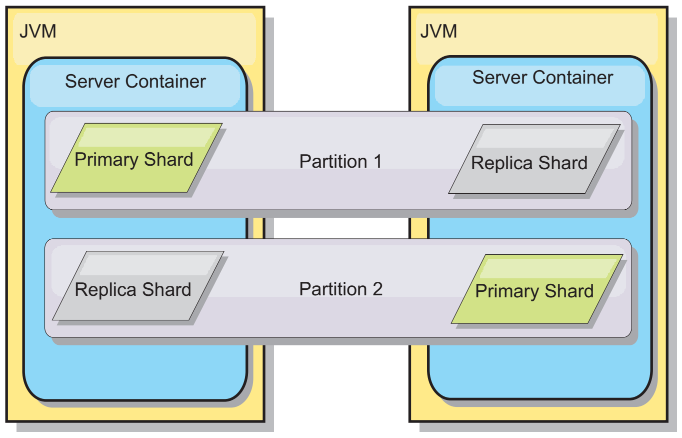


Figure 4. Partition

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 as the primary and are not normally hosted on the same machine as the primary.

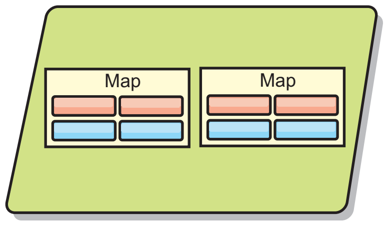


Figure 5. Shard

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.

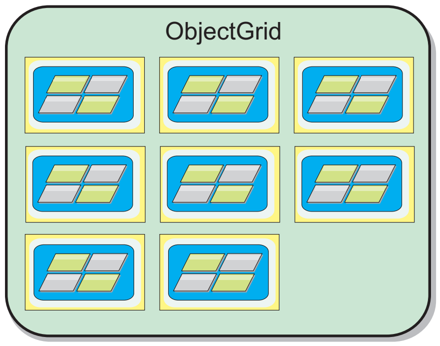


Figure 6. ObjectGrid

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.

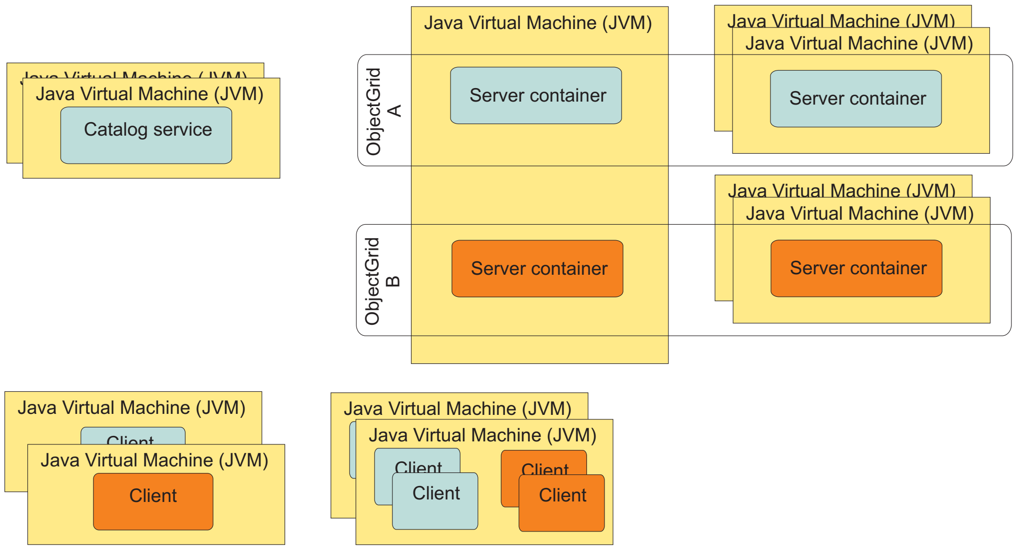

Java Virtual Machine (JVM)
Catalog service
Java Virtual Machine (JVM)
ObjectGrid A
Server container
ObjectGrid B
Server container
Java Virtual Machine (JVM)
Server container
Java Virtual Machine (JVM)
Server container
Java Virtual Machine (JVM)
Client
Java Virtual Machine (JVM)
Client
Client

Figure 7. 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.

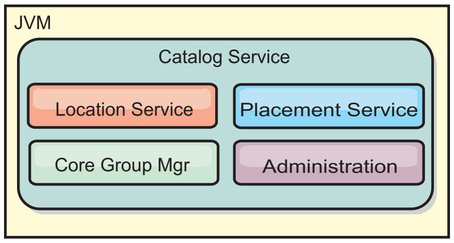


Figure 8. Catalog service

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 so 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. See the *Administration Guide* for more information on each stage.

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

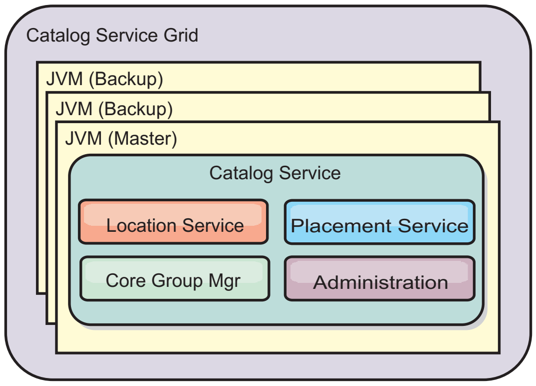


Figure 9. Catalog service grid

Local in-memory cache

In the simplest case, 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 eXtreme Scale data grid can be indexed and retrieved using WebSphere eXtreme Scale's query support. The ability to query the data can help developers greatly when working with large in memory data sets versus the limited data structure support provided with the Java virtual machine (JVM), which is ready to use as is.

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

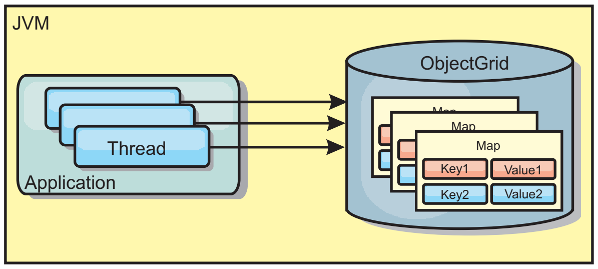


Figure 10. Local in-memory cache scenario

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 in-memory cache

For a local WebSphere eXtreme Scale cache, you must ensure the cache is synchronized if there are multiple processes with independent cache instances. To do so, enable a peer-replicated cache with 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 Java Messaging Service (JMS).

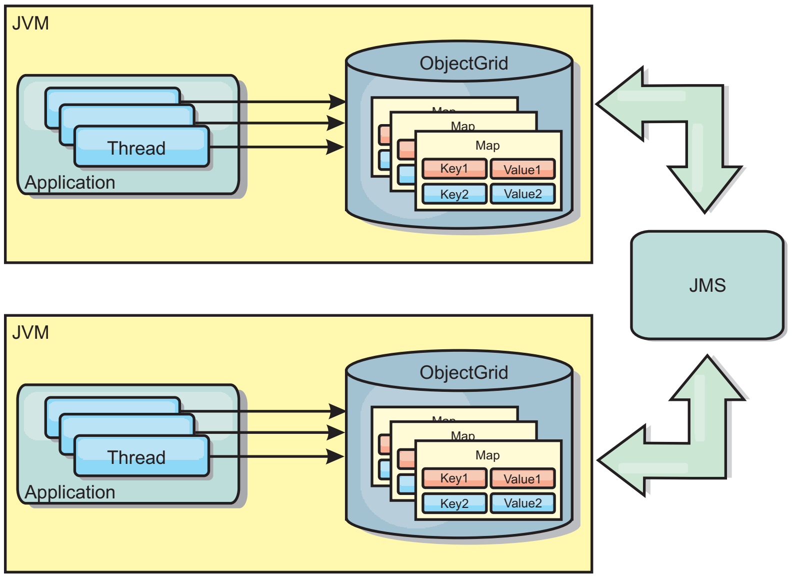


Figure 11. Peer-replicated cache with changes that are propagated with JMS

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 eXtreme Scale cache instance.

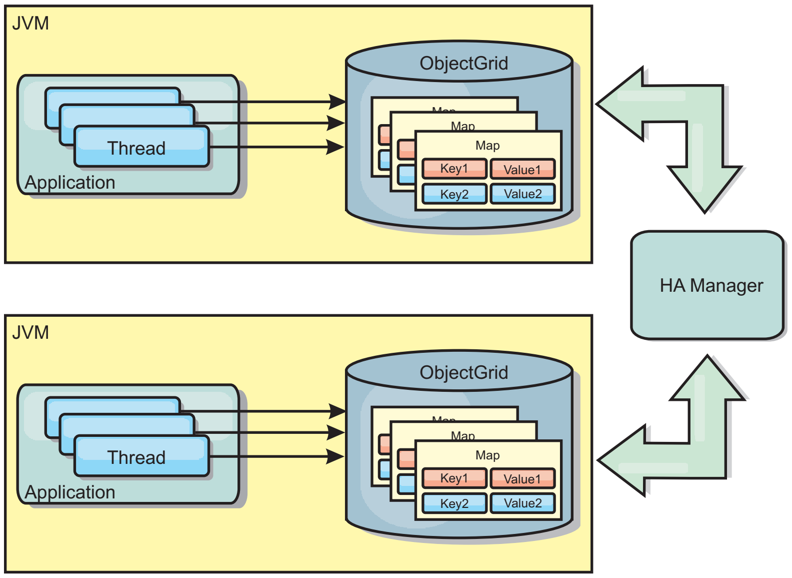


Figure 12. Peer-replicated cache with changes that are propagated with the high availability manager

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

This deployment topology should be used 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.

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 there is no stale data. 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.

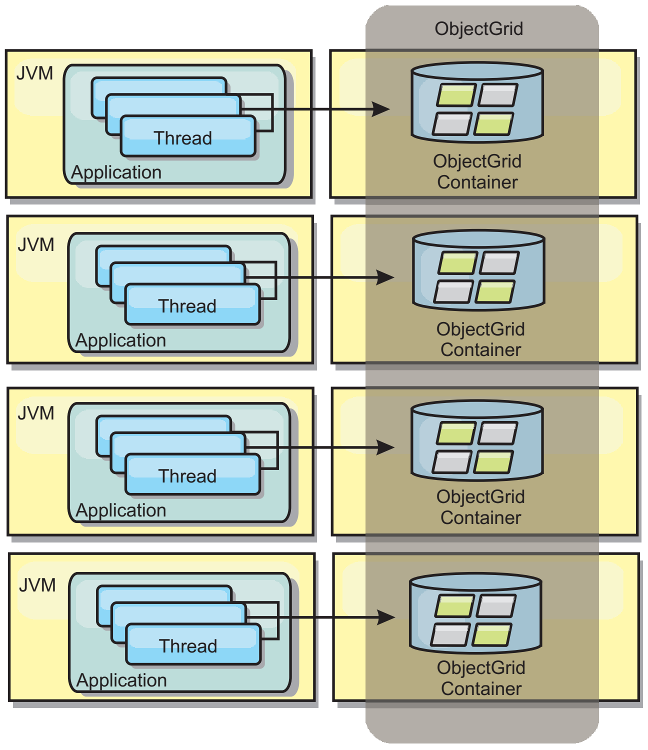


Figure 13. Distributed cache

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.

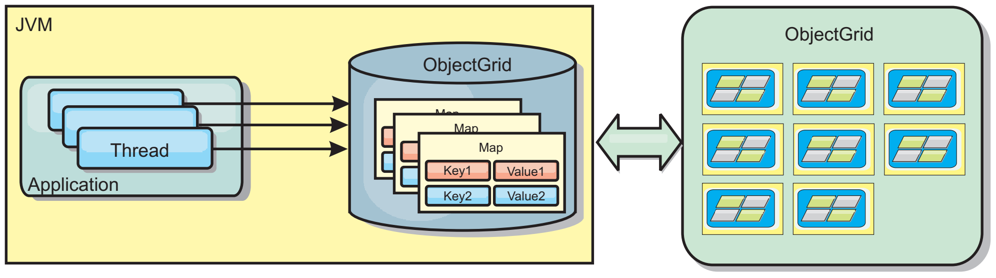


Figure 14. Near cache

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 near cache 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 data grid does not have the requested data, the Loader is invoked and the resulting data is inserted from the backend data store into the data 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.

Disadvantages

- Increases duration of stale data.
- Must use 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

eXtreme Scale data grids can run within existing processes as embedded eXtreme Scale servers or can be managed as external processes. Embedded data 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.

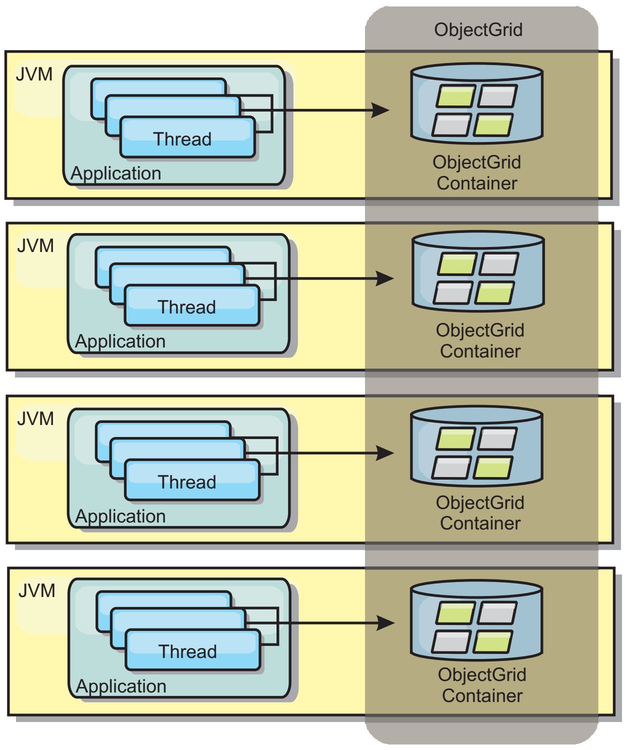


Figure 15. Embedded cache

Advantages

- Simplified administration since there are less processes to manage.
- Simplified application deployment since the data grid is using the client application's classloader.
- Support 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 data 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 data 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*.

Database integration

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.

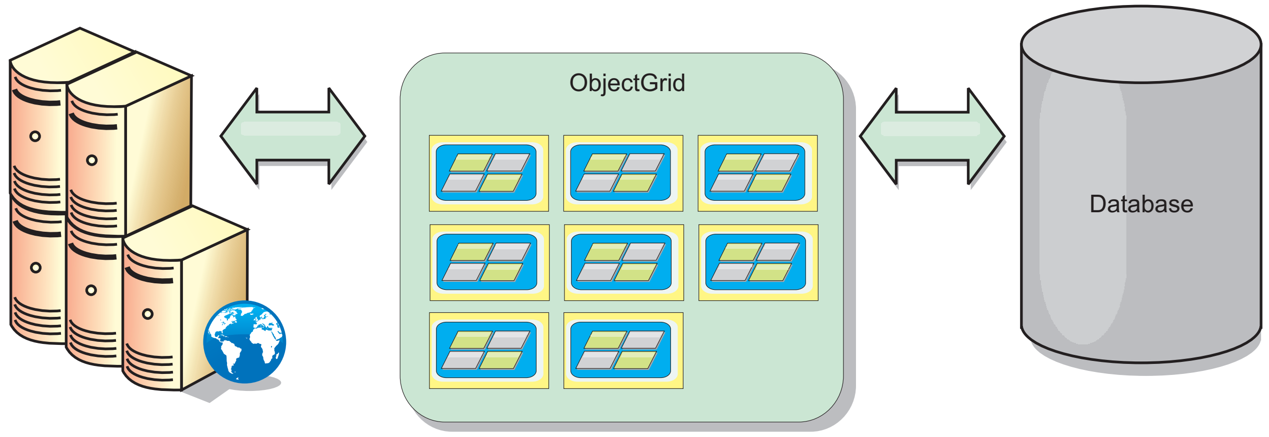


Figure 16. ObjectGrid as a database buffer

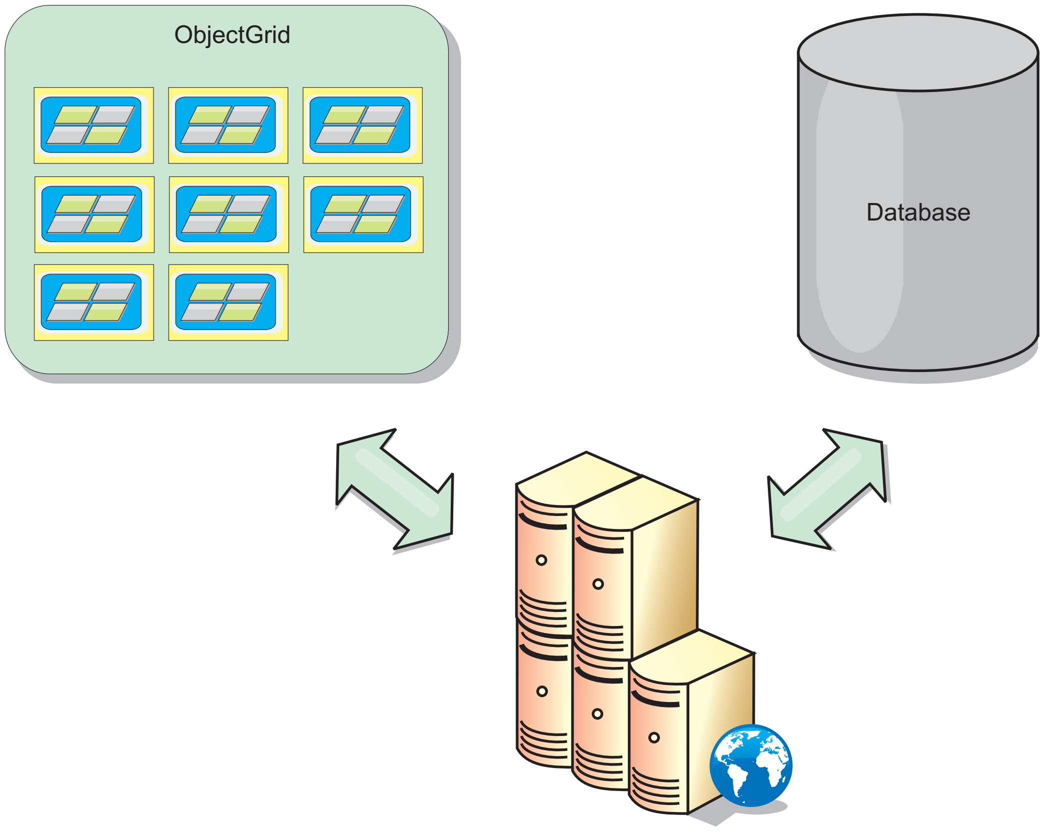


Figure 17. ObjectGrid as a side cache

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, on-demand. Sparse caches are normally accessed using keys (instead of indexes or queries) since the data is only partially available.

When a key is not present (a cache miss), the next tier is invoked and the data is fetched and inserted into the respective cache tier. If using a query or index, only the currently loaded values are accessed and the requests are not forwarded to the other tiers. 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 prior to applications' using it, and can effectively function as a database replacement. After data is loaded, it can be treated similarly to a database. Since all of the data is available, queries and indexes can be used to find and aggregate data.

Side cache and in-line cache

WebSphere eXtreme Scale is used to provide 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 a side cache, the back-end is used in conjunction with the eXtreme Scale.

Side cache

eXtreme Scale can be used as a side-cache for an application's data access layer. In this scenario, eXtreme Scale is used to temporarily store objects that would normally be retrieved from a back-end database. Applications check to see if eXtreme Scale contains the desired data. If the data is there, the data is returned to the caller. If the data is not there, the data is retrieved from the back-end and inserted into eXtreme Scale so that the next request can use the cached copy. The following diagram illustrates how eXtreme Scale can be used as a side-cache using an arbitrary data access layer such as OpenJPA or Hibernate.

Side cache plug-ins for Hibernate and OpenJPA

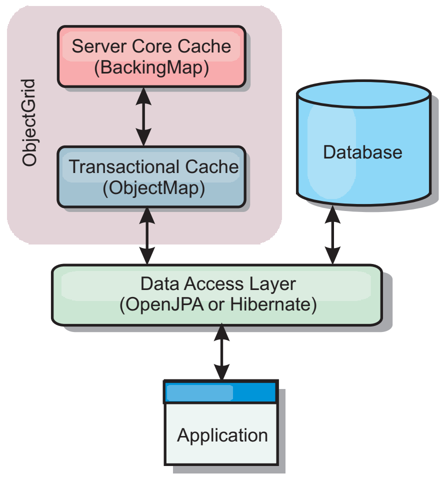

ObjectGrid
Server Core Cache (BackingMap)
Transactional Cache (ObjectMap)
Database
Data Access Layer (OpenJPA or Hibernate)
Application

Figure 18. Side cache

Cache plug-ins for both OpenJPA and Hibernate are included ineXtreme Scale , which allows you to use eXtreme Scale as an automatic side-cache. Using eXtreme Scale as a cache provider increases performance when reading and querying data and reduces load to the database. There are advantages thateXtreme 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 will be able to utilize the cached value.

In-line cache

When used as an in-line cache, eXtreme Scale interacts with the back-end using a Loader plug-in. This scenario can simplify data access by allowing applications to 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.

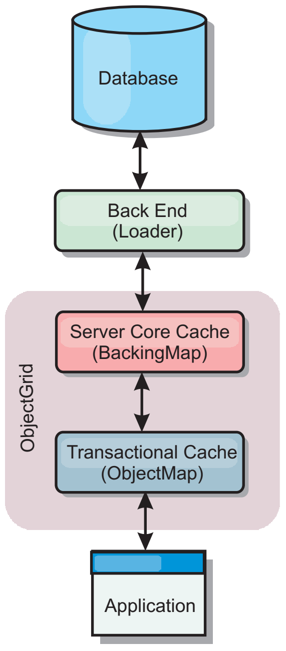


Figure 19. In-line cache

In-line caching scenarios

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 backend using a Loader plug-in.

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.

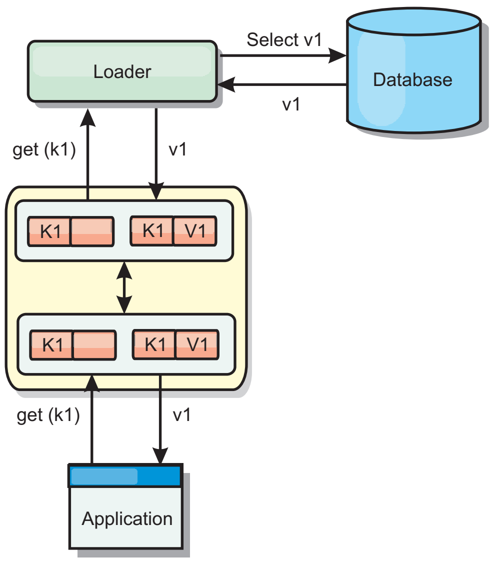


Figure 20. Read-through caching

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.

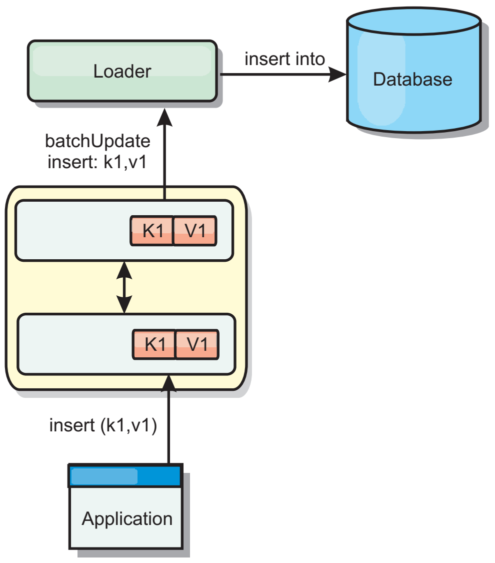


Figure 21. Write-through caching

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” on page 28 for more information.

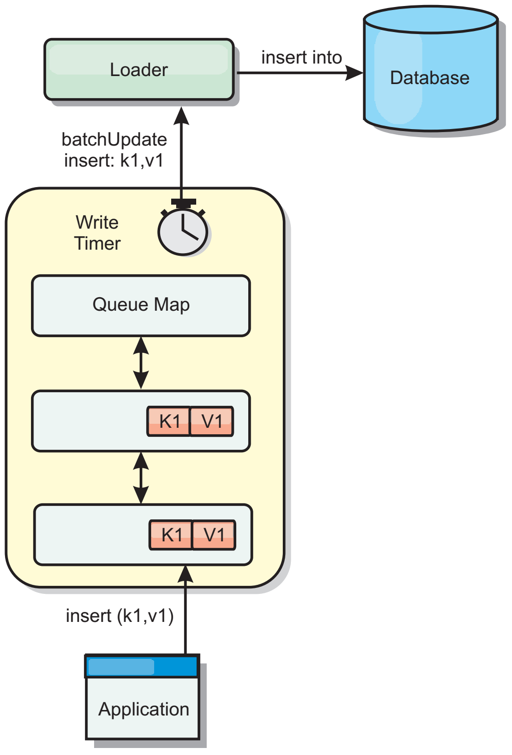


Figure 22. Write-behind caching

See “Write-behind caching” on page 28 for further information.

Loaders:

A Loader is a plug-in that works as a link between a BackingMap and a back-end such as a database.

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 numbe rof 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.

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 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 grid. Alternatively, a client preloader can be used to load the entire grid.

For more information about loaders, see the information about loaders in the *Administration 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 used a sparse cache, you should warm up the cache with data so that clients can have immediate access to data when they connect.

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

Loader plug-in

The loader plug-in is associated to each map and 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. So 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.

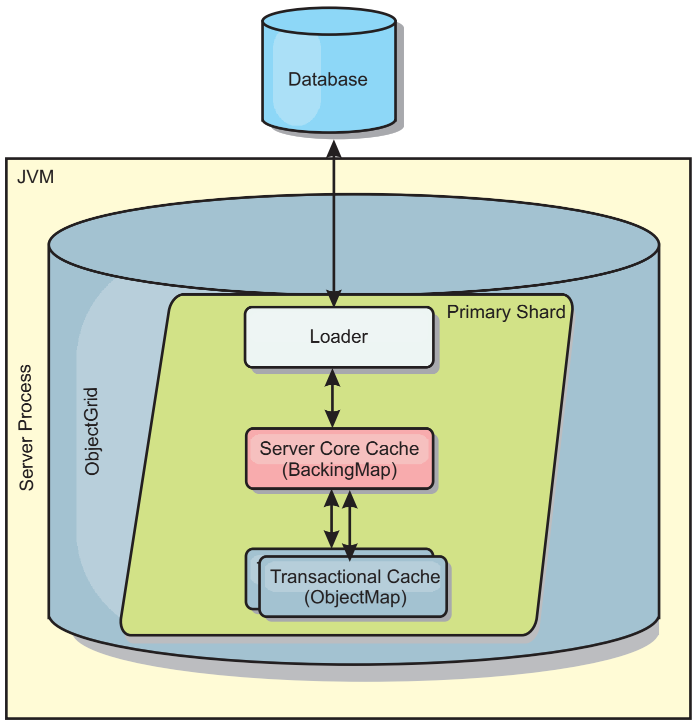

Database
JVM
Server Process
ObjectGrid
Primary Shard
Loader
Server Core Cache (BackingMap)
Transactional Cache (ObjectMap)

Figure 23. 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 data 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.

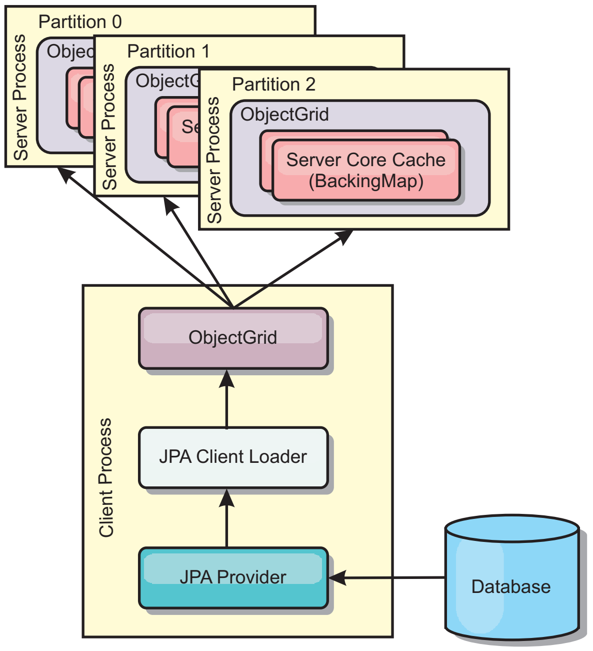


Figure 24. Client loader

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.

Introduction

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).

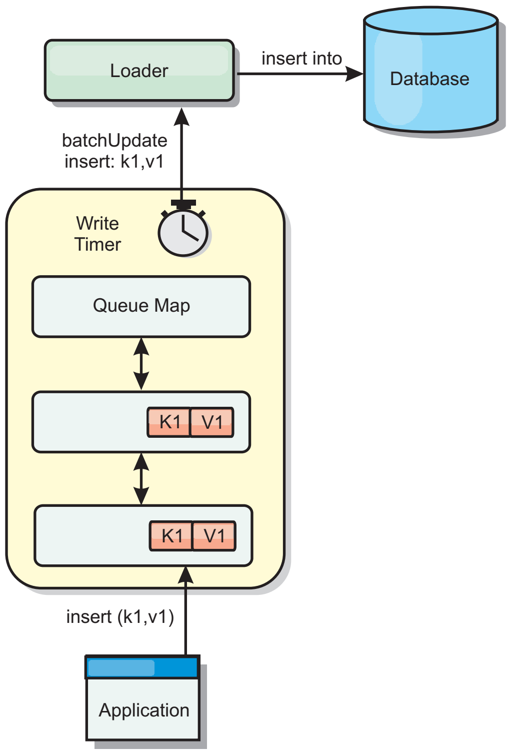


Figure 25. Write-behind caching

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 since 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.

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.

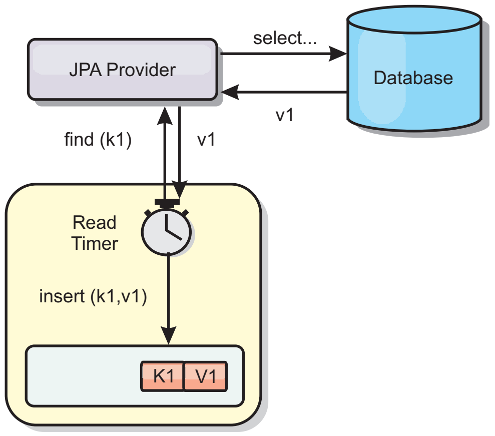


Figure 26. Periodic refresh

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. See the “Eviction” topic for further details.

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.

Eviction

WebSphere eXtreme Scale provides a default mechanism for evicting cache entries and a plug-in for creating custom evictors. An evictor controls the membership of entries in each BackingMap. The default evictor uses a time to live (TTL) eviction policy for each BackingMap. If you provide a pluggable evictor mechanism, it typically uses an eviction policy that is based on the number of entries instead of on time.

TimeToLive property

A default TTL evictor is created with every backing map. The default evictor removes entries based on a time to live concept. This behavior is defined by the ttlType attribute, which has three types.

- None: Specifies that entries never expire and therefore are never removed from the map.
- Creation time: Specifies that entries are evicted depending on when they were created.
- Last accessed time: Specifies that entries are evicted depending upon when they were last accessed.

If you are using the CREATION_TIME ttlType, the evictor evicts an entry when its time from creation equals its TimeToLive attribute value (which is set in milliseconds in your application configuration). If you set the TimeToLive attribute value to 10 seconds, the entry is automatically evicted ten seconds after it was inserted. It is important to take caution when setting this value for the CREATION_TIME ttlType. This evictor is best used when reasonably high amounts of additions to the cache exist that are only used for a set amount of time. With this strategy, anything that is created is removed after the set amount of time.

Following is an example of where a TTL type of CREATION_TIME is useful. You are using a Web application that obtains stock quotes, and getting the most recent quotes is not critical. In this case, the stock quotes are cached in an ObjectGrid for 20 minutes. After 20 minutes, the ObjectGrid map entries expire and are evicted. Every twenty minutes or so the ObjectGrid map uses the Loader plug-in to refresh the map data with fresh data from the database. The database is updated every 20 minutes with the most recent stock quotes. So for this application, using a TimeToLive value of 20 minutes is ideal.

If you are using the LAST_ACCESSED_TIME ttlType attribute, set the TimeToLive to a lower number than if you are using the CREATION_TIME ttlType, because the entries TimeToLive attribute is reset every time it is accessed. In other words, if the TimeToLive attribute is equal to 15 and an entry has existed for 14 seconds but then gets accessed, it does not expire again for another 15 seconds. If you set the TimeToLive to a relatively high number, many entries might never be evicted. However, if you set the value to something like 15 seconds, entries might be removed when they are not often accessed.

Following is an example of where a TTL type of LAST_ACCESSED_TIME is useful. An ObjectGrid map is used to hold session data from a client. Session data must be destroyed if the client does not use the session data for some period of time. For example, the session data times out after 30 minutes of no activity by the client. In this case, using a TTL type of LAST_ACCESSED_TIME with the TimeToLive attribute set to 30 minutes is exactly what is needed for this application.

The following example creates a backing map, set its default evictor ttlType attribute, and sets its TimeToLive property.

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");
bMap.setTtlEvictorType(TTLType.LAST_ACCESSED_TIME);
bMap.setTimeToLive(1800);
```

Most evictor settings should be set before you initialize the ObjectGrid.

You may also write your own evictors: For more information, see the information about writing a custom evictor in the *Programming Guide*.

Optional evictors

The default TTL evictor uses an eviction policy that is based on time, and the number of entries in the BackingMap has no affect on the expiration time of an entry. You can use an optional pluggable evictor to evict entries based on the number of entries that exist instead of based on time.

The following optional pluggable evictors provide some commonly used algorithms for deciding which entries to evict when a BackingMap grows beyond some size limit. *

- The LRUEvictor evictor uses a least recently used (LRU) algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.
- The LFUEvictor evictor uses a least frequently used (LFU) algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.

The BackingMap informs an evictor as entries are created, modified, or removed in a transaction. The BackingMap keeps track of these entries and chooses when to evict one or more entries from the BackingMap.

A BackingMap has no configuration information for a maximum size. Instead, evictor properties are set to control the evictor behavior. Both the LRUEvictor and the LFUEvictor have a maximum size property that is used to cause the evictor to begin to evict entries after the maximum size is exceeded. Like the TTL evictor, the LRU and LFU evictors might not immediately evict an entry when the maximum number of entries is reached to minimize impact on performance.

If the LRU or LFU eviction algorithm is not adequate for a particular application, you can write your own evictors to create your eviction strategy.

Memory-based eviction

Important: Memory-based eviction is only supported on Java Platform, Enterprise Edition Version 5 or later.

All built-in evictors support memory-based eviction that can be enabled on the BackingMap interface by setting the evictionTriggers attribute of BackingMap to `MEMORY_USAGE_THRESHOLD`. For more information about how to set the evictionTriggers attribute on BackingMap, see the information about the BackingMap interface and the ObjectGrid descriptor XML file in the *Administration Guide*.

Memory-based eviction is based on heap usage threshold. When memory-based eviction is enabled on BackingMap and the BackingMap has any built-in evictor, the usage threshold is set to a default percentage of total memory if the threshold has not been previously set.

When you are using memory-based eviction, you should configure the garbage collection threshold to the same value as their target heap utilization. For example, if the memory-based eviction threshold is set at 50 percent and the garbage collection threshold is at the default 70 percent level, then the heap utilization can go as high as 70 percent. This heap utilization increase occurs because memory-based eviction is only triggered after a garbage collection cycle.

The memory-based eviction algorithm used by WebSphere eXtreme Scale is sensitive to the behavior of the garbage collection algorithm in use. The best algorithm for memory-based eviction is the IBM default throughput collector. Generation garbage collection algorithms can cause undesired behavior, and so you should not use these algorithms with memory-based eviction.

To change the usage threshold percentage, set the memoryThresholdPercentage property on the container and server property files for eXtreme Scale server processes.

For more information, see details about configuring the property files, starting stand-alone servers, and starting container processes in the *Administration Guide*.

During runtime, if the memory usage exceeds the target usage threshold, memory-based evictors start evicting entries and try to keep memory usage below the target usage threshold. However, no guarantee exists that the eviction speed is fast enough to avoid a potential out of memory error if the system runtime continues to quickly consume memory.

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 "Architecture and topology" on page 9 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 using relationships in a data grid, the data must be organized in a constrained tree. There must be one root type 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. This is a special case where the data should exist in every partition. Here, the data can be accessed by any root key and the same result will be returned. Reference data such as this should only be used in cases where the data is fairly static since updating it can be expensive, since it 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 data 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 data 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.

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 is queryable 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

WebSphere eXtreme Scale can integrate with other caching-related products. JPA can be used between WebSphere eXtreme Scale and the database to integrate changes as a loader. You can also use the WebSphere eXtreme Scale dynamic cache provider to plug WebSphere eXtreme Scale into the dynamic cache component in WebSphere Application Server. Another extension to WebSphere Application Server is the WebSphere eXtreme Scale HTTP session manager, which can help to cache HTTP sessions.

Using eXtreme Scale with JPA

The Java Persistence API (JPA) is a specification that allows mapping Java objects to relational databases. JPA contains a full object-relational mapping (ORM) specification using Java language metadata annotations, XML descriptors, or both to define the mapping between Java objects and a relational database. A number of open-source and commercial implementations are available.

To use JPA, you must have a supported JPA provider, such as OpenJPA or Hibernate, JAR files, and a `META-INF/persistence.xml` file in your class path.

JPA Loaders overview

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.

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).

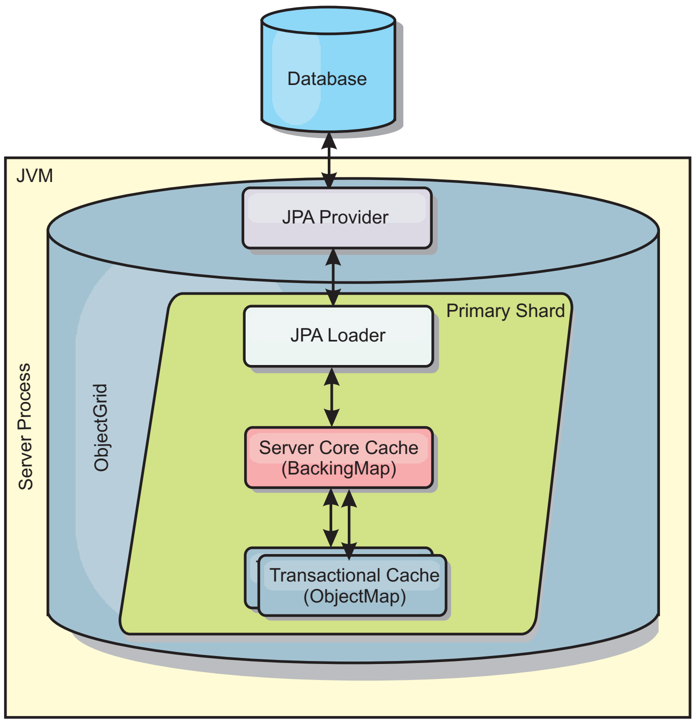


Figure 27. JPA Loader architecture

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.

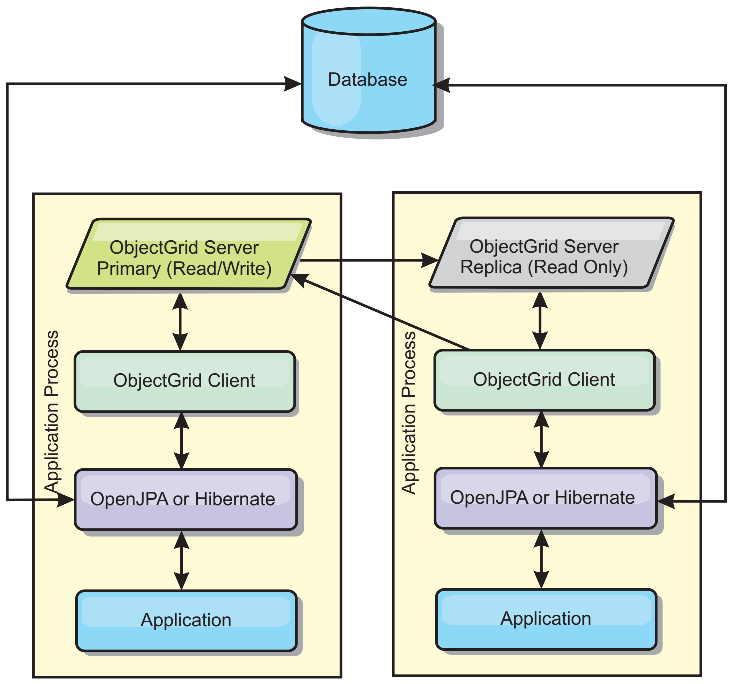


Figure 28. JPA embedded topology

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.

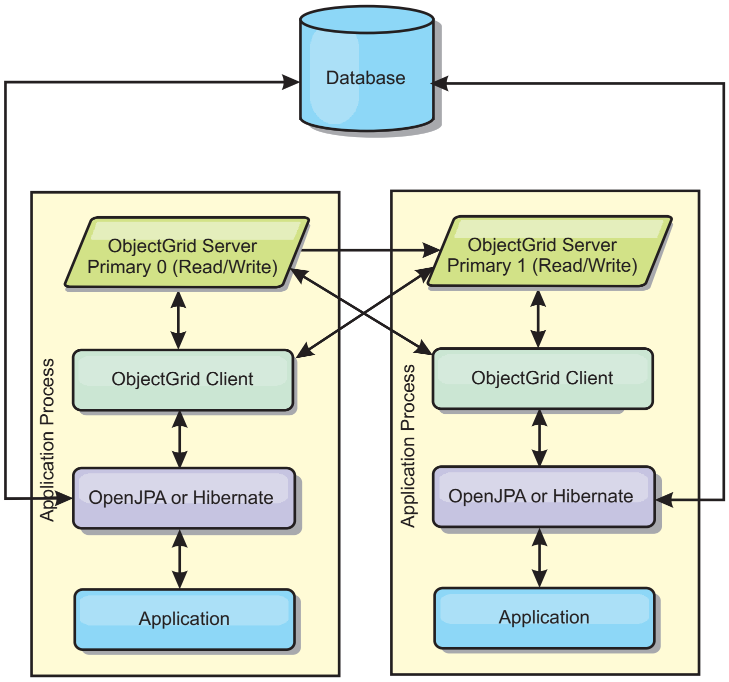


Figure 29. JPA embedded, partitioned topology

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.

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 configure the eXtreme Scale to be partitioned and replicated. A remote configuration is managed independently from the application and JPA provider.

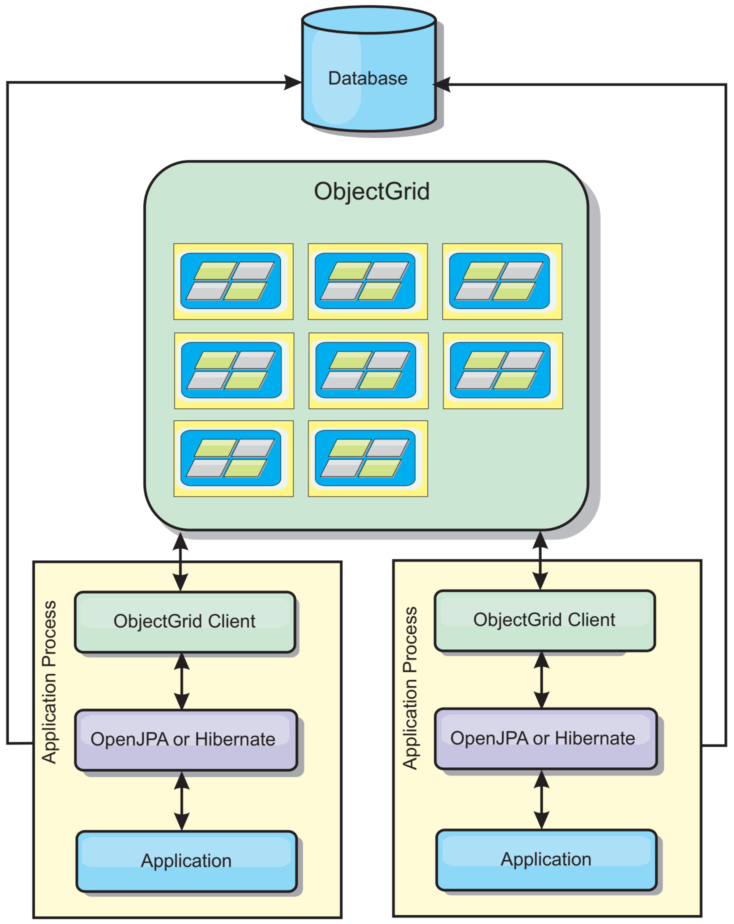


Figure 30. JPA remote topology

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

In a WebSphere Application Server Version 5 or later environment, the HTTP session manager that is shipped with WebSphere eXtreme Scale can override the default session manager in the application server.

The WebSphere eXtreme Scale HTTP session manager can also run on WebSphere Application Server Version 6.0.2 or later or in an environment that is not running WebSphere Application Server, such as WebSphere Application Server Community Edition or Apache Tomcat.

Features

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 management capabilities for an associated application. The session manager creates HTTP sessions and manages 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.

How the session manager works

The session manager introduces itself into the request path in the form of a servlet filter. You can add this servlet filter to every Web module in your application with tooling that ships with WebSphere eXtreme Scale. You can also manually add these filters to the Web deployment descriptor of your application. This filter receives the request before the servlet or JSP files in the target application. At this time, the filter intercepts the HTTPRequest and HTTPResponse objects and creates the respective wrapper objects for its own implementation.

This filter implementation intercepts all HTTP session related calls that are made on the HTTPRequest and HTTPResponse objects. These calls are satisfied by the session manager and are not passed through to the base session manager in the underlying JEE server implementation. The session manager creates and maintains sessions and their life cycles, including timeout-based invalidations and the firing of listeners on session invalidations and other life cycle events.

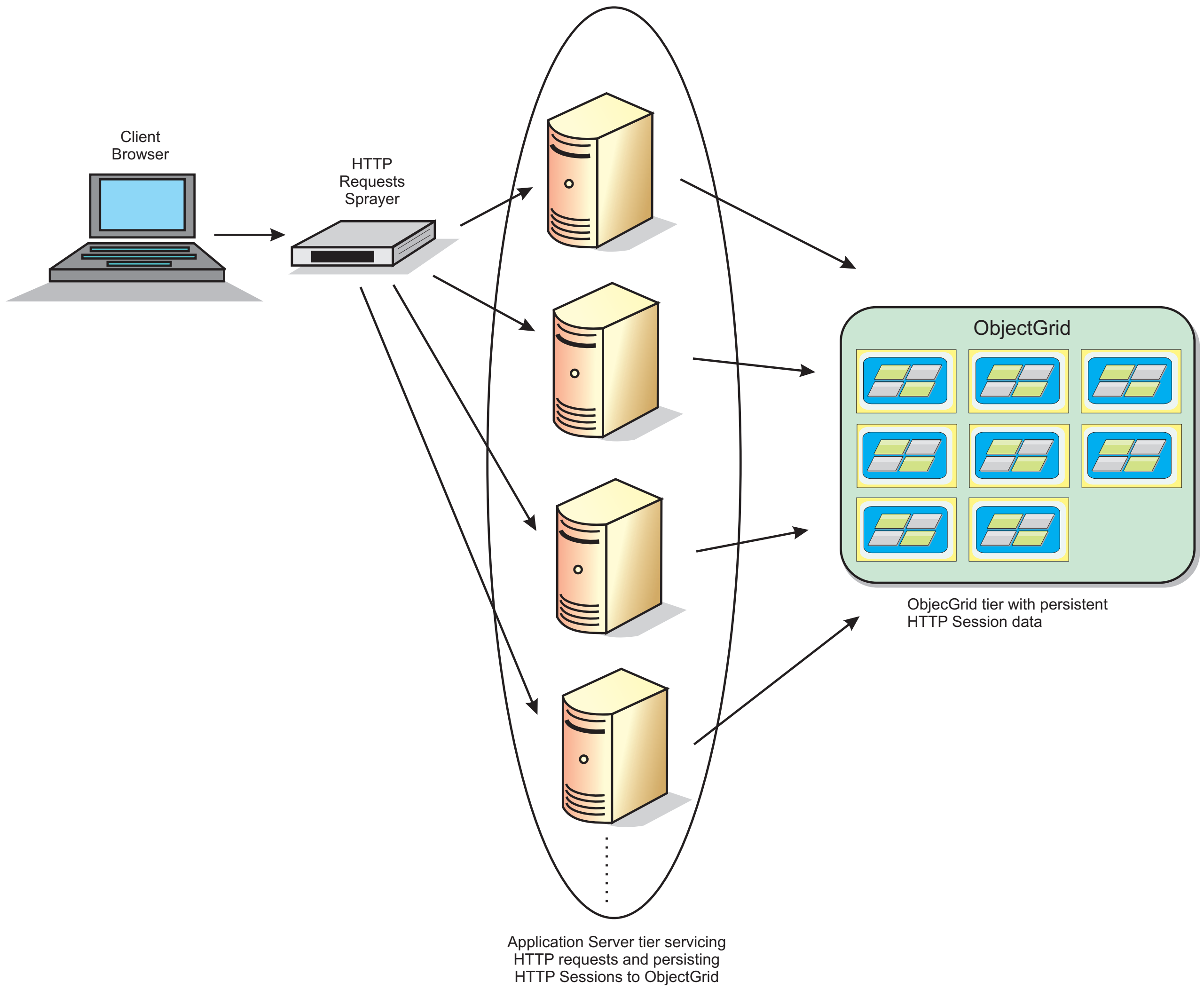

Client Browser
HTTP Requests Sprayer
ObjectGrid
ObjecGrid tier with persistent HTTP Session data
Application Server tier servicing HTTP requests and persisting HTTP Sessions to ObjectGrid

Figure 31. 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.
- **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.

Retrieving an eXtreme Scale session in a session manager application

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

        HttpSession session = req.getSession(true);

        System.out.println("calling getSession");
        // call getAttribute("com.ibm.websphere.objectgrid.session")
    // to get the ObjectGrid session instance
        Session ogSession = (Session)session.getAttribute
      ("com.ibm.websphere.objectgrid.session");

        System.out.println("ogSession = "+ogSession);
    }
```

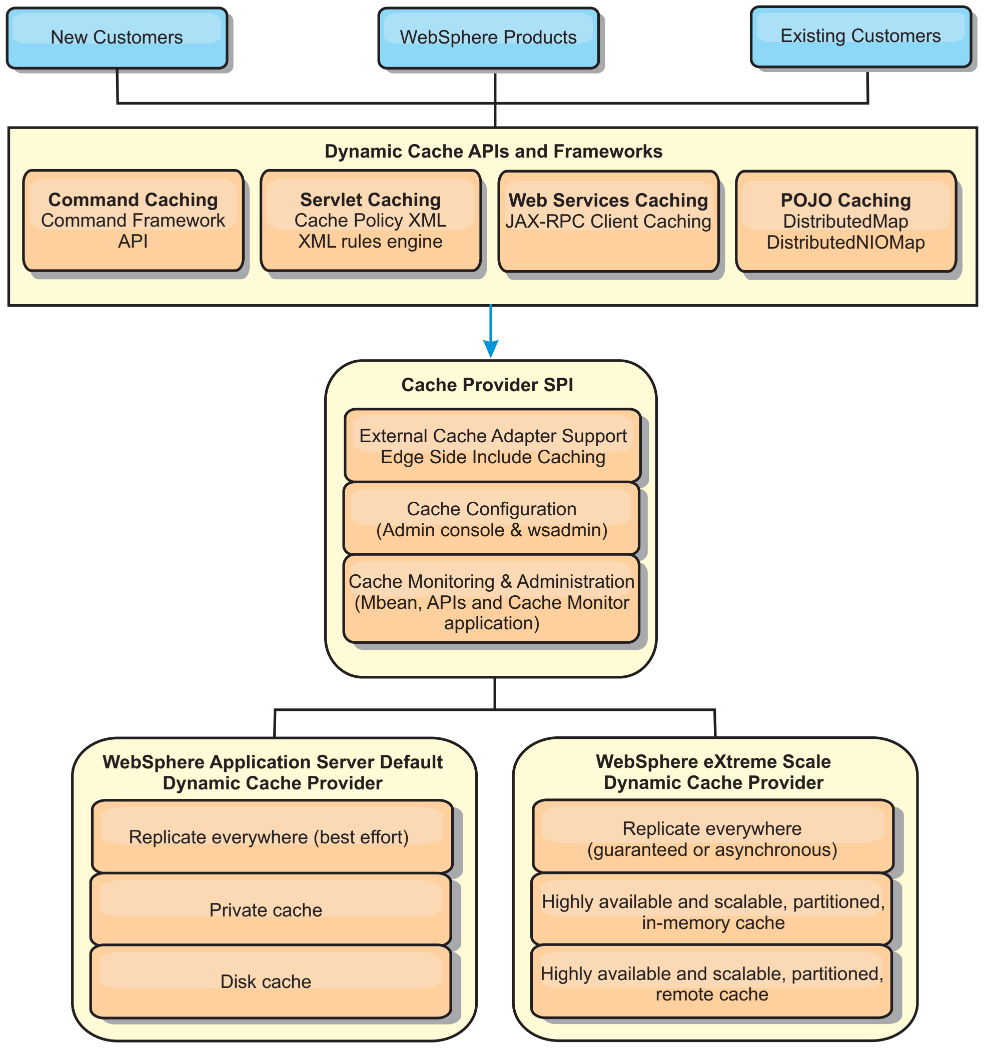
Any changes made to the maps with the session that is returned from the getAttribute method call are committed when the underlying session is committed. See the servlet context initialization parameters in the *Administration Guide* for more information.

WebSphere eXtreme Scale dynamic cache provider

The IBM DynaCache 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

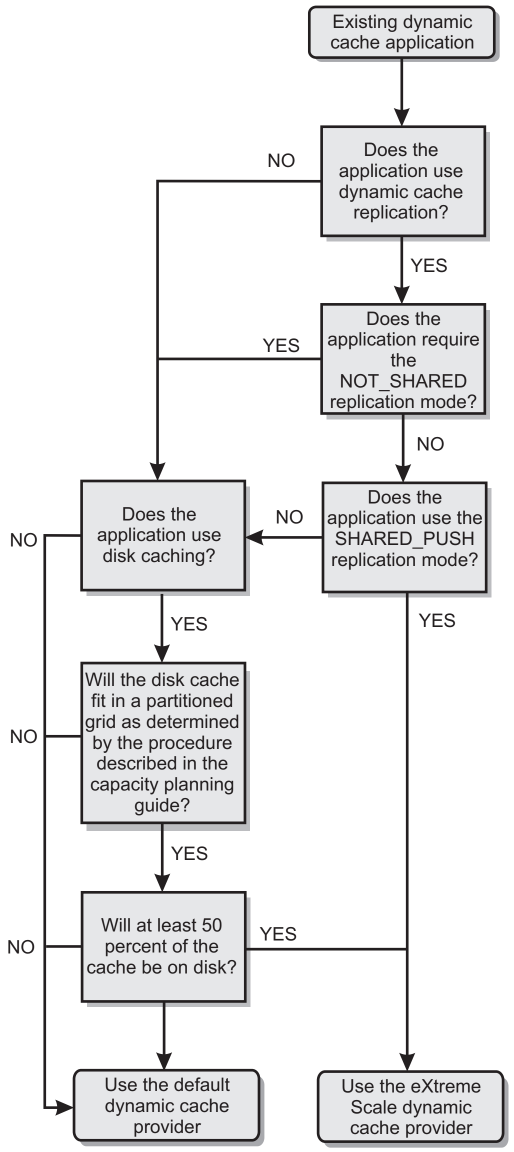
You can install and configure the dynamic cache provider as described in Configuring the dynamic cache provider for WebSphere eXtreme Scale.

Deciding how to leverage WebSphere eXtreme Scale

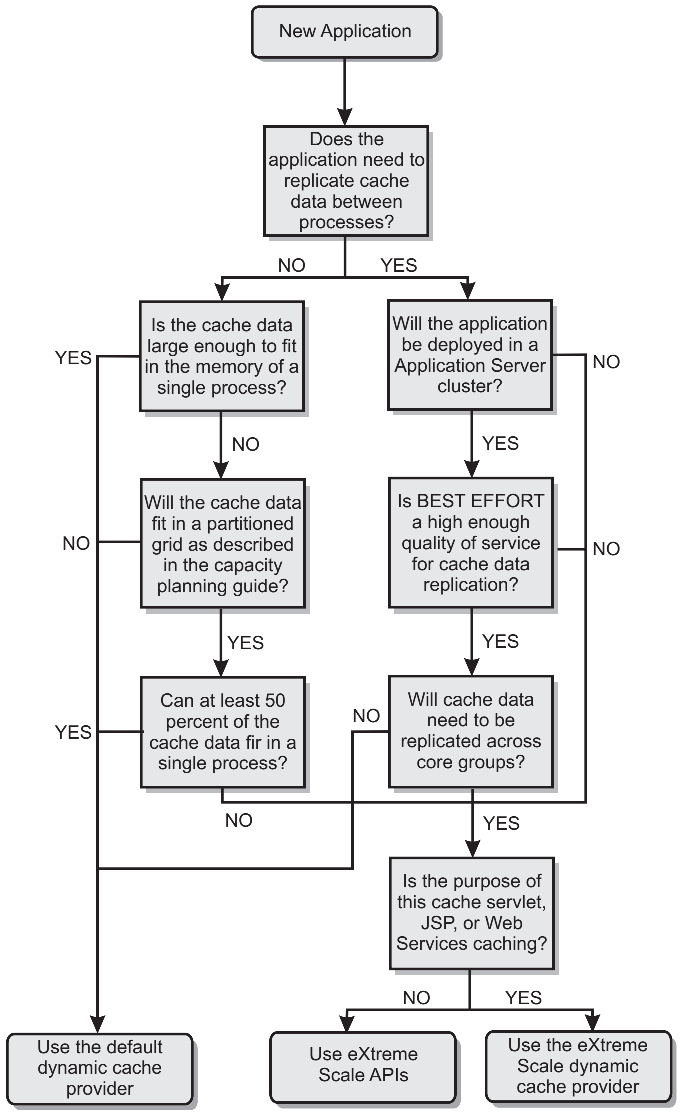
The available features in WebSphere eXtreme Scale significantly increase the distributed capabilities of the dynamic cache API beyond what is offered by the default dynamic cache engine and data replication service. With eXtreme Scale, you can create caches that are truly distributed between multiple 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 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?
Will the application be deployed in a Application Server cluster?
YES
NO
NO
YES
Will the cache data fit in a partitioned grid as described in the capacity planning guide?
Is BEST EFFORT a high enough quality of service for cache data replication?
NO
NO
YES
YES
Can at least 50 percent of the cache data fir in a single process?
Will cache data need to be replicated across core groups?
YES
NO
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 3. 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		

Table 3. Feature comparison (continued)

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
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
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 4. 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 5. Programming interfaces

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale API
Command-based API	Command framework API	Command framework API	DataGrid API

Table 5. Programming interfaces (continued)

Cache features	Default provider	eXtreme Scale provider	eXtreme Scale 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 "Deployment configurations for eXtreme Scale" in the *Programming Guide*.

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

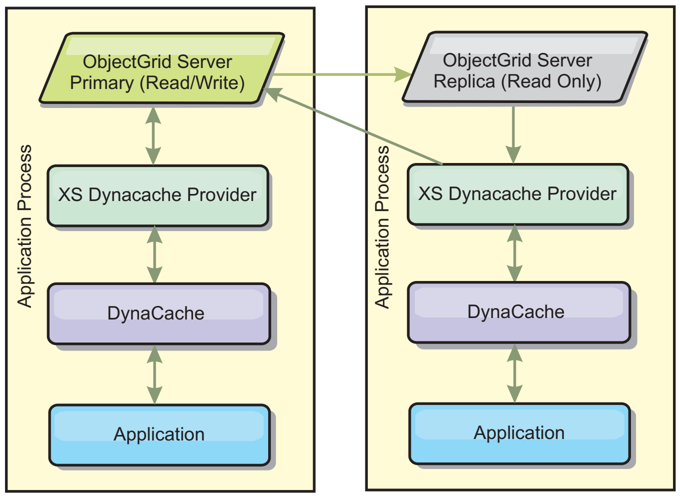
Topology types

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

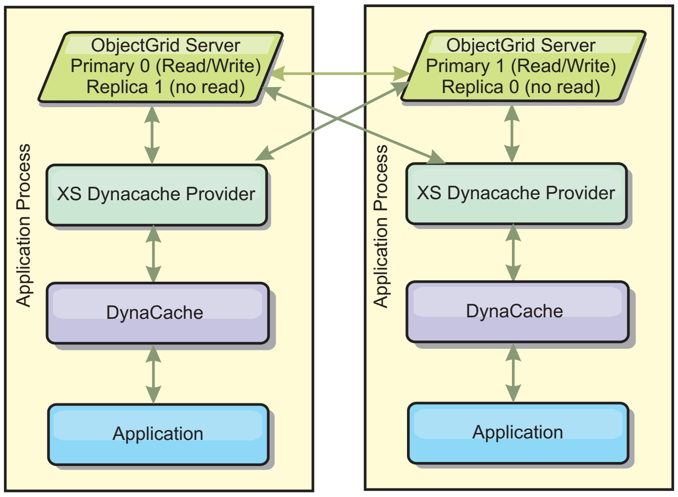
The embedded topology is similar to the default dynamic cache and DRS provider. Distributed cache instances created with the embedded topology keep a full copy of the cache within each eXtreme Scale process that accesses the dynamic cache service, allowing all read operations to occur locally. All write 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

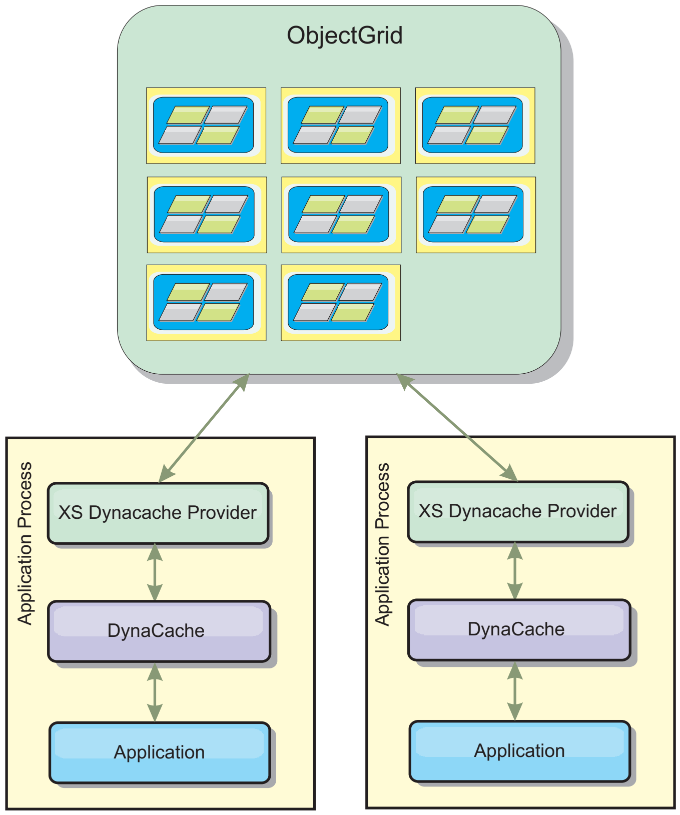
For workloads where cache-writes occur as often as or more frequently than reads, the embedded partitioned or remote topologies are recommended. The embedded partitioned topology keeps all of the cache data within the WebSphere Application Server processes that access the cache. However, each process 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 not currently supported and will always return -1.
- **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 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

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 machines at run time. You can, for example, start with a deployment of four servers and then expand to a deployment with ten servers as the demands on the cache grow. Just as you can add more physical machines and processing units for vertical scalability, you may extend eXtreme Scale's elastic scaling capability horizontally with partitioning. This is another major difference with in-memory databases (IMDBs) as opposed to eXtreme Scale (which is a data grid), since IMDBs can only scale vertically.

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

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 to 20. Only peer to peer replication between shards exists.

Partitioning

Partitioning is the mechanism that WebSphere eXtreme Scale uses to scale out an application. Partitioning is the separation of the application state into parts where each part contains some set of complete instance data. 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, so eXtreme Scale does not currently support two-phase commit across partitions.

Important: Select your number of partitions carefully. The number of partitions that are defined in the deployment policy directly affects the number of containers 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.

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.

Partitioning

Use partitioning to store large amounts of data in the Java Virtual Machine (JVM). To partition data, use an application-specified scheme to divide the data. With eXtreme Scale, partitioning increases both scalability and availability.

Using partitions

A grid can have many partitions, or thousands of partitions if required. A 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 grid of four container Java virtual machines. The application should be configured to have a reasonable number of shards for each container 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 computers with two container Java virtual machines per computer. You expect to grow to 20 computers over the next three years. With 20 computers, you have 40 container Java virtual machines, and choose 60 to be pessimistic. You want 4 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.

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. All child entities are assumed to be in the same partition as the root. The child entities in this 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, since 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 element. 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 placementStrategy 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.

For more information, see the topic on the per-container placement strategy in the *Programming Guide*. .

Per-container placement strategy

As opposed to a fixed-partition placement policy, you may also enable per-container placement, which allows for different configurations that are not possible in a fixed-partition scenario.

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 grid. A 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 per-container grid

PER_CONTAINER grids are different. You specify that the grid use PER_CONTAINER through the placementPolicy attribute in your deployment XML file. Instead of configuring how many partitions total you want in the grid, you specify how many partitions you want per container that you start.

For example, if you set the number of partitions per container to be 5, then when you start a container eXtreme Scale creates 5 new anonymous partition primaries on that container and creates any necessary replicas on the other containers already deployed.

The following is a potential sequence in a per-container environment as the 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 5 new primary partitions each time and rebalancing replicas on the available containers in the grid.

Note: WebSphere eXtreme Scale does not move primaries 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 5 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 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 5 partitions per container, then 5 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 grid that does not rely on keys. The client just begins a transaction and stores data in the 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 grid.

Advantages

The previous description is different from a normal FIXED_PARTITION or hash 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 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 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).

PartitionableKey interface

When an eXtreme Scale configuration uses the fixed partition placement strategy, it depends on hashing the key to a partition to insert, get, update, or remove the value. The hashCode method is called on the key and it must be well defined if a custom key is created. However another option is to use the PartitionableKey interface. With the PartitionableKey interface, you can use an object other than the key to hash to a partition.

You can use the PartitionableKey interface in situations where there are multiple maps and the data you commit is related and thus should be put on the same partition. WebSphere eXtreme Scale does not support two-phase commit so multiple map transactions should not be committed if they span multiple partitions. If the PartitionableKey hashes to the same partition for keys in different maps in the same map set, they can be committed together.

You can also use the PartitionableKey interface when groups of keys should be put on the same partition, but not necessarily during a single transaction. If keys should be hashed based on location, department, domain type, or some other type of identifier, children keys can be given a parent PartitionableKey.

For example, employees should hash to the same partition as their department. Each employee key would have a PartitionableKey object that belongs to the department map. Then both the employee and department would hash to the same partition.

The PartitionableKey interface supplies one method, called ibmGetPartition. The object returned from this method must implement the hashCode method. The result returned from using the alternate hashCode will be used to route the key to a partition.

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 5. Availability

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

WebSphere eXtreme Scale self-organizes grids of Java virtual machines into a loosely federated tree, with the catalog service at the root and core groups holding containers at the leaves of the tree. See the "Architecture and topology" on page 9 topic 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.

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 very 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.

eXtreme Scale container failure

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 of 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 failover 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, very quickly. Typical failure detection for soft failures is less than one second. Hard failures may take up to 200 seconds to detect using the default heart beat tuning. Such failures include: physical machine crashes, network cable disconnects or operating system failures. Thus, eXtreme Scale must rely on heart beating to detect hard failures which can be configured. See "Failover detection types" on page 108 for details on lowering the time it takes to detect a hard failure.

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 grid uses a peer election process for defining the primary shard instead of the catalog service algorithm that is used for the containers.

Note that the placement service and the core grouping service are one-of-N services, but the location service and administration run everywhere. 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 should restart to restore the system to the desired level of availability. If all of the processes that are hosting the catalog service fail, eXtreme Scale has a loss of critical data. This failure results in a required restart of all the containers. 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. The deployment policy defines a development mode boolean attribute that the catalog service uses to determine whether a replica can be placed on the same machine as a primary. In a development environment on a single machine, you might want to have two containers and replicate between them. However, in production, using a single machine is insufficient because loss of that host results in the loss of both containers. 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.

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 62 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 6. 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 7. 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 8. 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.

Replication architecture

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. An asynchronous replica still maintains the order of the transactions that are sent from the primary.

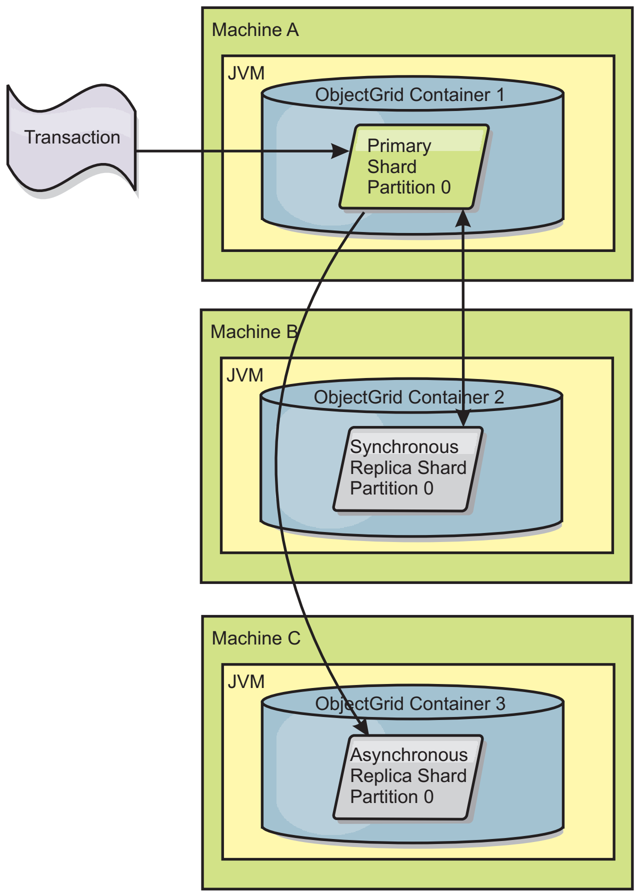


Figure 32. Communication path between a primary shard and replica shards

The memory cost of replication

To bring a replica online, a checkpoint map must be created to copy over existing data from the primary. However, the main memory cost on the primary is the memory for each entry that is modified after the checkpoint occurs. If the data changes a lot after the checkpoint, then this operation costs an amount of memory that is proportional to the number of modified entries. After the checkpoint is copied to the replica, the changes are merged back in to the checkpoint. The changed entries are not freed until they have been sent to all required replicas.

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 allocation: primary and replica

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 data 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.

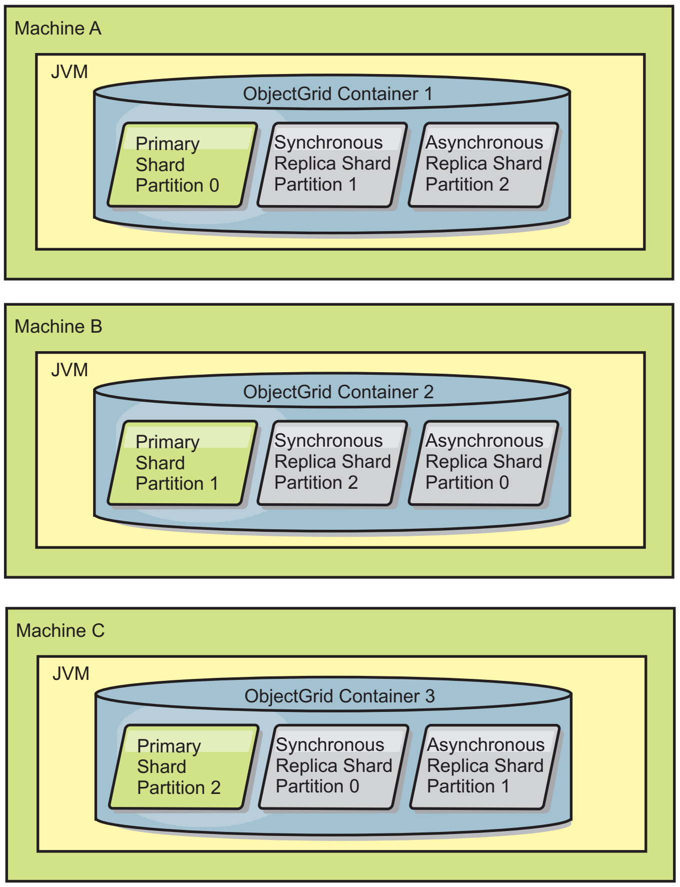

Machine A
JVM
ObjectGrid Container 1
Primary Shard Partition 0
Synchronous Replica Shard Partition 1
Asynchronous Replica Shard Partition 2
Machine B
JVM
ObjectGrid Container 2
Primary Shard Partition 1
Synchronous Replica Shard Partition 2
Asynchronous Replica Shard Partition 0
Machine C
JVM
ObjectGrid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

Figure 33. 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

High-availability catalog service

A catalog service is the 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 data grid for high availability.

Components of the catalog service

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 server grid is propagated to the clients through the Common Object Request Broker Architecture (CORBA) service context.

Configure at least three catalog servers. 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 server grid 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 server grid and the container server grid are very different. The catalog server grid is for high availability of your system data. The container grid is for your data high availability, scalability, and workload management. Therefore, two different routing tables exist: the routing table for the catalog server grid and the routing table for the server grid shards.

The catalog responsibilities are divided into a series of services. The core group manager performs peer grouping for health monitoring, the placement service performs allocation, the administration service provides access to administration, and the location service manages locality.

Catalog grid deployment

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 placement of shards across the set of available containers. The placement service is responsible for maintaining a balance of resources across physical resources. The placement service is responsible for allocating individual shards to their host container. It runs as a one-of-N elected service in the grid, so there is always exactly one instance of the service 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 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 containers that are registering hosted applications with the placement service. The location service runs on all of the grid members to scale out this function.

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 grid.

Planning

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

Starting a catalog server grid

See the details about starting a catalog service grid in the *Administration Guide* for more information.

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

You can configure eXtreme Scale containers that are embedded in a WebSphere Application Server environment to connect to a stand-alone catalog grid. Use the same property to connect the application server to the catalog grid. However, the property does not manage the life cycle of the catalog with the servers. Instead, the property allows the containers to locate the remote catalog grid. To set the property, see the details about starting a catalog service in a WebSphere Application Server environment in the *Administration Guide* for more information.

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:

Quorum is the minimum number of catalog servers necessary to conduct placement operations for the data grid. The minimum number is the full set of catalog servers unless quorum has been overridden.

Important terms

The following is a list of terms related to quorum considerations for eXtreme Scale.
- Brown out: A brown out is the temporary loss of connectivity between one or more servers.
- Black out: A black out is the permanent loss of connectivity between one or more servers.

- Data center: A data center is a geographically located group of servers generally connected with a local area network (LAN).
- Zone: A zone is a configuration option used to group servers together that share some physical characteristic. The following are some examples of zones for a group of servers: a data center as described in the previous bullet, an area network, a building, a floor of a building, and so on.
- Heartbeat: Heartbeating is a mechanism used to determine whether or not a given Java virtual machine (JVM) is running.

Topology

This section explains how IBM WebSphere eXtreme Scale operates across a network that includes unreliable components. Examples of such a network would include a network spanning multiple data centers.

IP address space

WebSphere eXtreme Scale requires a network where any addressable element on the network can connect to any other addressable element on the network unimpeded. This means WebSphere eXtreme Scale requires a flat IP address naming space and it requires all firewalls to all traffic to flow between the IP addresses and ports used by the Java virtual machines (JVM) hosting elements of WebSphere eXtreme Scale.

Connected LANs

Each LAN is assigned a zone identifier for WebSphere eXtreme Scale requirements. WebSphere eXtreme Scale aggressively heartbeats JVMs in a single zone and a missed heartbeat will result in a failover event if the catalog service has quorum. Read about Configuring failover detection for more information.

Catalog service data grid and container servers

A data grid is a collection of similar JVMs. A catalog service is a data grid composed of catalog servers, and is fixed in size. However, the number of container servers is dynamic. Container servers can be added and removed on demand. In a three-data-center configuration, WebSphere eXtreme Scale requires one catalog service JVM per data center.

The catalog service data grid uses a full quorum mechanism. This means that all members of the data grid must agree on any action.

Container server JVMs are tagged with a zone identifier. The data grid of container JVMs is automatically broken in to small core groups of JVMs. A core group will only include JVMs from the same zone. JVMs from different zones will never be in the same core group.

A core group will aggressively try to detect the failure of its member JVMs. The container JVMs of a core group must never span multiple LANs connected with links like in a wide area network. This means that a core group cannot have containers in the same zone running in different data centers.

Server life cycle

Catalog server startup

The catalog servers are started using the startOgServer command. The quorum mechanism is disabled by default. To enable quorum either pass the `-quorum` enabled flag on the startOgServer command or add the `enableQuorum=true` property in the property file. All of the catalog servers must be given the same quorum setting.

```
# bin/startOgServer cat0 –serverProps objectGridServer.properties
```

objectGridServer.properties file

```
catalogClusterEndPoints=cat0:cat0.domain.com:6600:6601,
cat1:cat1.domain.com:6600:6601
catalogServiceEndPoints= cat0.domain.com:2809, cat1.domain.com:2809
enableQuorum=true
```

Container server startup

The container servers are started using the startOgServer command. When running a data grid across data centers the servers must use the zone tag to identify the data center in which they reside. Setting the zone on the data grid servers allows WebSphere eXtreme Scale to monitor health of the servers scoped to the data center, minimizing cross-data-center traffic.

```
# bin/startOgServer gridA0 –serverProps objectGridServer.properties –
objectgridfile xml/objectgrid.xml –deploymentpolicyfile xml/
deploymentpolicy.xml
```

objectGridServer.properties file

```
catalogServiceEndPoints= cat0.domain.com:2809, cat1.domain.com:2809
zoneName=ZoneA
```

Data grid server shutdown

The grid servers are stopped using the stopOgServer command. When shutting down an entire data center for maintenance, pass the list of all the servers that belong to that zone. This will allow a clean transition of state from the zone in teardown to the surviving zone or zones.

```
# bin/stopOgServer gridA0,gridA1,gridA2 –catalogServiceEndPoints
cat0.domain.com:2809,cat1.domain.com:2809
```

Failure detection

WebSphere eXtreme Scale detects process death through abnormal socket closure events. The catalog service will be told immediately when a process terminates. A black out is detected through missed heartbeats. WebSphere eXtreme Scale protects itself against brown out conditions across data centers by using a quorum implementation.

Heartbeating implementation

This section describes how liveness checking is implemented in WebSphere eXtreme Scale.

Core group member heartbeating

The catalog service places container JVMs into core groups of a limited size. A core group will try to detect the failure of its members using two methods. If a JVM's socket is closed, that JVM is regarded as dead. Each 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 period of time then the JVM is regarded as dead.

A single member of a core group is always elected to be the leader. The core group leader (CGL) is responsible to periodically tell the catalog service that the core group is alive and to report any membership changes to the catalog service. A membership change can be a JVM failing or a newly added JVM joining the core group.

If the core group leader cannot contact any member of the catalog service grid then it will continue to retry.

Catalog service grid heartbeating

The catalog service 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 heartbeating configuration.

Core group heartbeating

The catalog service needs to know when container servers fail. Each core group is responsible for determining container JVM failure and reporting this to the catalog service through the core group leader. 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.

If the catalog service marks a container JVM as failed and the container is later reported as alive, the container JVM will be told to shutdown the WebSphere eXtreme Scale container servers. A JVM in this state will not be visible in xsadmin queries. There will be messages in the logs of the container JVM indicating this has happened. These JVMs need to be manually restarted.

If a quorum loss event has occurred, heartbeating is suspended until quorum is reestablished.

Catalog service quorum behavior

Normally, the members of the catalog service have full connectivity. The catalog service grid is a static set of JVMs. WebSphere eXtreme Scale expects all members of the catalog service to be online always. The catalog service will only respond to container events while the catalog service has quorum.

If the catalog service loses quorum, it will wait for quorum to be reestablished. While the catalog service does not have quorum, it will ignore events from container servers. Container servers will retry any requests rejected by the catalog server during this time as WebSphere eXtreme Scale expects quorum to be reestablished.

The following message indicates that quorum has been lost. Look for this message in your catalog service logs.

```
CWOBJ1254W: The catalog service is waiting for quorum.
```

WebSphere eXtreme Scale expects to lose quorum for the following reasons:

- Catalog service JVM member fails
- Network brown out
- Data center loss

Stopping a catalog server instance using `stopOgServer` does not cause loss of quorum because the system knows the server instance has stopped, which is different from a JVM failure or brown out.

Quorum loss from JVM failure

A catalog server that fails will cause quorum to be lost. In this case, quorum should be overridden as fast as possible. The failed catalog service cannot rejoin the grid until quorum has been overridden.

Quorum loss from network brown out

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

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

Catalog service JVM cycling

If a catalog server is stopped by using stopOgServer, then the quorum drops to one less server. This means the remaining servers still have quorum. Restarting the catalog server bumps quorum back to the previous number.

Consequences of lost quorum

If a container JVM was to fail while quorum is lost, recovery will not take place until the brown out recovers or in the case of a black out the customer does an override quorum command. WebSphere eXtreme Scale regards a quorum loss event and a container failure as a double failure, which is a rare event. This means that applications may lose write access to data that was stored on the failed JVM until quorum is restored at which time normal recovery will take place.

Similarly, if you attempt to start a container during a quorum loss event, the container will 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 brown out occurs then some clients may not have access to primary or replica copies of the data until the brown out clears.

New clients can be started, as there should be a catalog service JVM in each data center so at least one catalog service JVM can be reached by a client even during a brown out event.

Quorum recovery

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

Container behavior

This section describes how the container server JVMs behave while quorum is lost and recovered.

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 will honor that assignment until new instructions arrive from the catalog service. This means that if a primary shard in a container cannot communicate with a replica shard because of a brown out then it will continue to retry until it receives new instructions from the catalog service.

If a network brown out occurs and a primary shard loses communication with the replica then it will retry the connection until the catalog service provides new instructions.

Synchronous replica behavior

While the connection is broken the primary can accept new transactions as long as there are at least as many replicas online as the minsync property for the map set. If any new transactions are processed on the primary while the link to the synchronous replica is broken, the replica will be cleared and resynchronized with the current state of the primary when the link is reestablished.

Synchronous replication is strongly discouraged between data centers or over a WAN-style link.

Asynchronous replica behavior

While the connection is broken the primary can accept new transactions. The primary will buffer 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

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

Quorum commands with xsadmin

This section describes xsadmin commands useful for quorum situations.

Querying quorum status

The quorum status of a catalog server instance can be interrogated using the xsadmin command.

```
xsadmin –ch cathost –p 1099 –quorumstatus
```

There are five possible outcomes.

- Quorum is disabled: The catalog servers are running in a quorum-disabled mode. This is a development or single only data center mode. It is not recommended for multi-data-center configurations.
- Quorum is enabled and the catalog server has quorum: Quorum is enabled and the system is working normally.
- Quorum is enabled but the catalog server is waiting for quorum: Quorum is enabled and quorum has been lost.
- Quorum is enabled and the quorum is overridden: Quorum is enabled and quorum has been overridden.
- Quorum status is outlawed: When a brown out occurs, splitting the catalog service into two partitions, A and B. The catalog server A has overridden quorum. The network partition resolves and the server in the B partition is outlawed, requiring a JVM restart. It also occurs if the catalog JVM in B restarts during the brown out and then the brown out clears.

Overriding quorum

The xsadmin command can be used to override quorum. Any surviving catalog server instance can be used. All survivors are notified when one is told to override quorum. The syntax for this is as follows.

```
xsadmin –ch cathost –p 1099 –overridequorum
```

Diagnostic commands

- Quorum status: As described in the previous section.
- Coregroup list: This displays a list of all core groups. The members and leaders of the core groups are displayed.
  ```
  xsadmin –ch cathost –p 1099 –coregroups
  ```
- Teardown servers: This command removes a server manually from the grid. This is not needed normally since servers are automatically removed when they are detected as failed, but the command is provided for use under IBM support help.
  ```
  xsadmin –ch cathost –p 1099 –g Grid –teardown server1,server2,server3
  ```
- Display route table: This command shows the current route table by simulating a new client connection to the grid. It also validates the route table by confirming that all container servers are recognizing their role in the route table, such as which type of shard for which partition.

```
xsadmin –ch cathost –p 1099 –g myGrid -routetable
```

- Display unassigned shards: If some shards cannot be placed on the grid then this can be used to list them. This only happens when the placement service has a constraint preventing placement. For example, if you start JVMs on a single physical box while in production mode then only primary shards can be placed. Replicas will be unassigned until JVMs start on a second box. The placement service only places replicas on JVMs with different IP addresses than the JVMs hosting the primary shards. Having no JVMs in a zone can also cause shards to be unassigned.

  ```
  xsadmin –ch cathost –p 1099 –g myGrid –unassigned
  ```

- Set trace settings: This command sets the trace settings for all JVMs matching the filter specified for the xsadmin command. This setting only changes the trace settings until another command is used or the JVMs modified fail or stop.

  ```
  xsadmin –ch cathost –p 1099 –g myGrid –fh host1 –settracespec
  ObjectGrid*=event=enabled
  ```

  This enables trace for all JVMs on the box with the host name specified, in this case host1.

- Checking map sizes: The map sizes command is useful for verifying that key distribution is uniform over the shards in the key. If some containers have significantly more keys than others then it is likely the hash function on the key objects has a poor distribution.

  ```
  xsadmin -ch cathost -p 1099 -g myGrid -m myMapSet -mapsizes myMap
  ```

Overriding quorum

This should only be used when a data center failure has occurred. Quorum loss due to a catalog service JVM failure or a network brownout should recovery automatically once the catalog service JVM is restarted or the network brownout clears.

Administrators are the only ones with knowledge of a data-center failure. WebSphere eXtreme Scale treats a brownout and a blackout similarly. You are required to inform the eXtreme Scale environment of any outages using the xsadmin command to override quorum which causes the catalog service to register that quorum is achieved with the current membership. Having achieved quorum allows your deployment to fully recover. Issuing an override quorum command confirms that JVMs in the failed data-center have failed beyond recovery.

The following list considers some scenarios for overriding quorum. Assume that you have deployed three catalog servers: A, B, and C.

- Brown out: Say you have a brown out where C is isolated temporarily. The catalog service will lose quorum and wait for the brown out to clear at which point C rejoins the catalog service grid and quorum is reestablished. Your application sees no problems during this time.
- Temporary failure: Here, C fails and the catalog service loses quorum, so you must override quorum. Once quorum is reestablished, C can be restarted. C will rejoin the catalog service grid when it is restarted. The application is unaffected during a temporary failure.
- Data center failure: You verify that the data center actually failed and that it has been isolated on the network. Then you issue the xsadmin override quorum command. The two surviving data centers fully recover by replacing shards that were hosted in the failed data center. The catalog service is now running with a full quorum of A and B. The application may see delays or exceptions during

the interval between the start of the black out and when quorum is overridden. Once quorum is overridden the grid recovers and normal operation is resumed.

- Data center recovery: The surviving data centers are already running with quorum overridden. When the data center containing C is restarted, all JVMs in the data center must be restarted. Then C will rejoin the existing catalog service grid and quorum will revert to the normal situation with no user intervention.
- Data center failure and brown out: The datacenter containing C fails. Quorum is overridden and recovered on the remaining data centers. If a brown out between A and B occurs, the normal brown out recovery rules apply. Once the brown out clears, quorum is reestablished and necessary recovery from the quorum loss occurs.

Transport security considerations

Since data centers are normally deployed in different geographical locations, users might want to enable transport security between the data centers for security reasons.

Read about transport layer security in the *Administration Guide*.

Life cycle, recovery, and failure events

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 allocation: primary and replica” on page 82 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.
```

When the 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.

To get a copy of the data on the primary shard, both synchronous and asynchronous replica shards perform the same startup sequence. During the registration of the replica shard, the primary shard creates a checkpoint of the current data. The data on the primary shard is in a copy-on-write state. The current data that is going to the replica shard is never modified, but a copy-on-write is performed whenever a new transaction updates records on the primary. The primary shard is able to continue processing without changing the data that is going to the replica shard. The primary shard keeps a list of the changes that were made since the checkpoint.

The checkpoint data is pushed to the replica. After the checkpoint arrives at the replica, the memory for the checkpoint is released. The changes since the checkpoint was created are merged. The list of changes are also pushed to the replica shard. As the changes are pushed across, the memory for the change is released.

After the replica completes the checkpoint phase, it switches into peer mode and begins to receive data as the primary receives the data. At this point, the replica shard starts to behave as either a synchronous or an asynchronous replica shard.

When a replica shard reaches peer mode, it prints a message to the `SystemOut.log` file for the replica.

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

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, including the checkpoint data and any additional changes that are made during the checkpoint copy. The time might display as zero or very low if the primary shard does not have any existing data to replicate.

Synchronous replica shard

If the new replica is a synchronous replica shard, the primary shard now starts a request or response whenever a transaction commits on the primary. The primary shard waits until the replica shard responds that it got the data. The synchronous replica shard data remains at the same level as the primary shard data.

Asynchronous replica shard

If the new replica is an asynchronous replica shard, the primary shard sends the data to the replica, but it does not wait for a response. The asynchronous replica orders and applies the data sent from the primary. The asynchronous replica data shard is not guaranteed to remain at the same level as the primary shard data.

Peer mode for all replica shards

When you are in peer mode, after all replica shards receive a transaction change, then the memory for the transaction is released on the primary shard. The replica shards only receive data from the primary shard during transactions that change data such as inserts, updates and removes. They replica shards are not contacted for reading data on the primary shard.

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. 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. If an asynchronous replica shard was behind, it receives a fresh copy of the data and register.

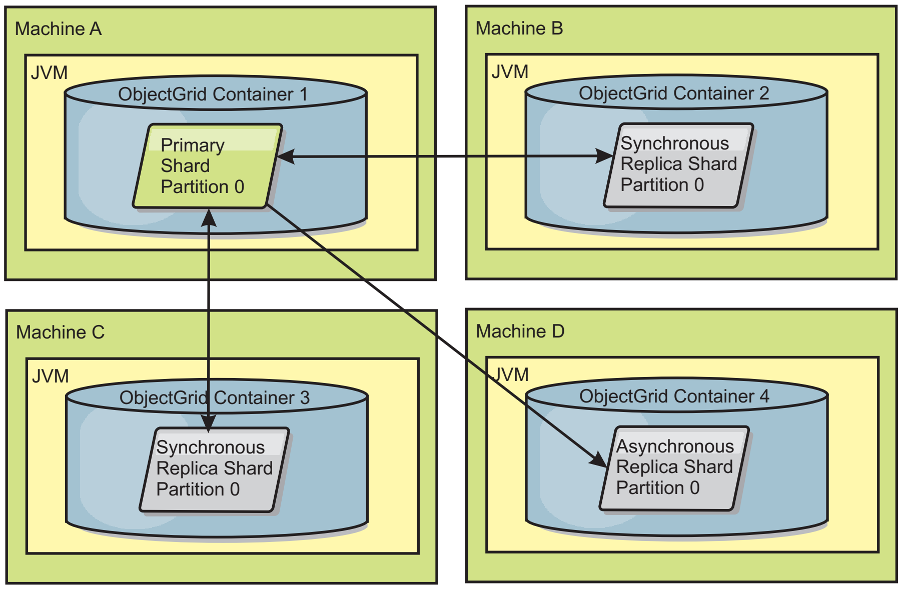

Machine A
JVM
ObjectGrid Container 1
Primary Shard Partition 0
Machine B
JVM
ObjectGrid Container 2
Synchronous Replica Shard Partition 0
Machine C
JVM
ObjectGrid Container 3
Synchronous Replica Shard Partition 0
Machine D
JVM
ObjectGrid Container 4
Asynchronous Replica Shard Partition 0

Figure 34. 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.

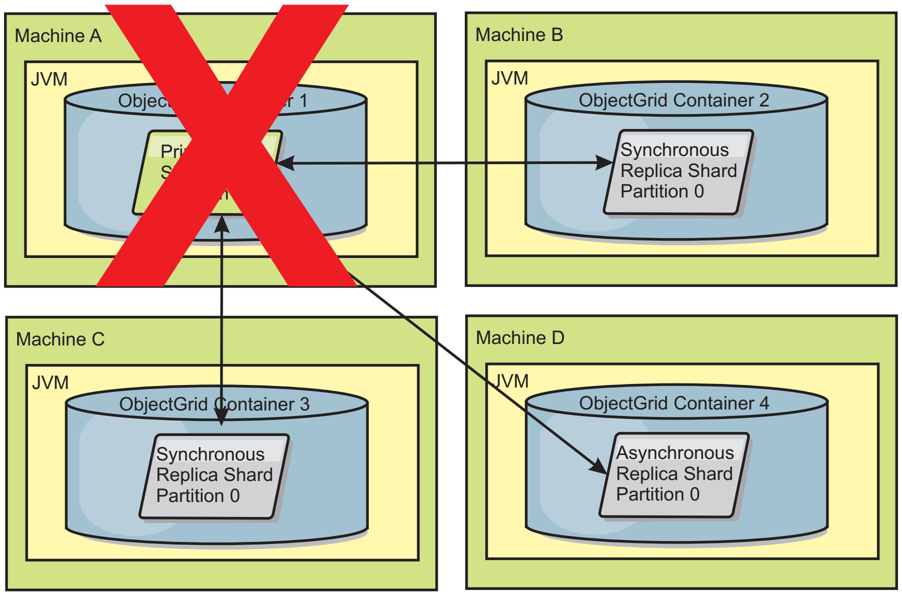


Figure 35. The container for the primary shard fails

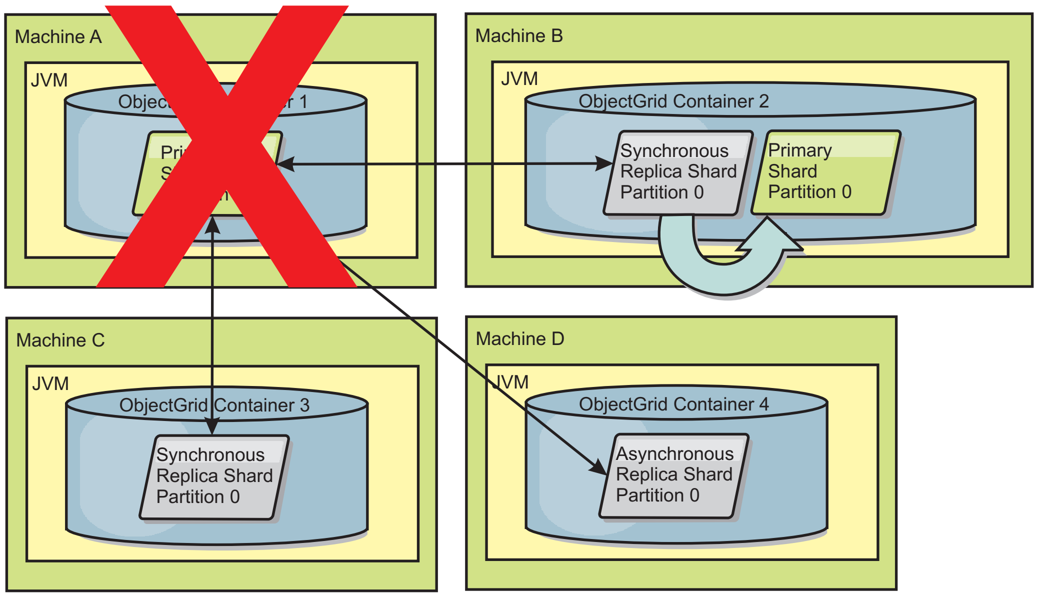


Figure 36. The synchronous replica shard on ObjectGrid container 2 becomes the primary shard

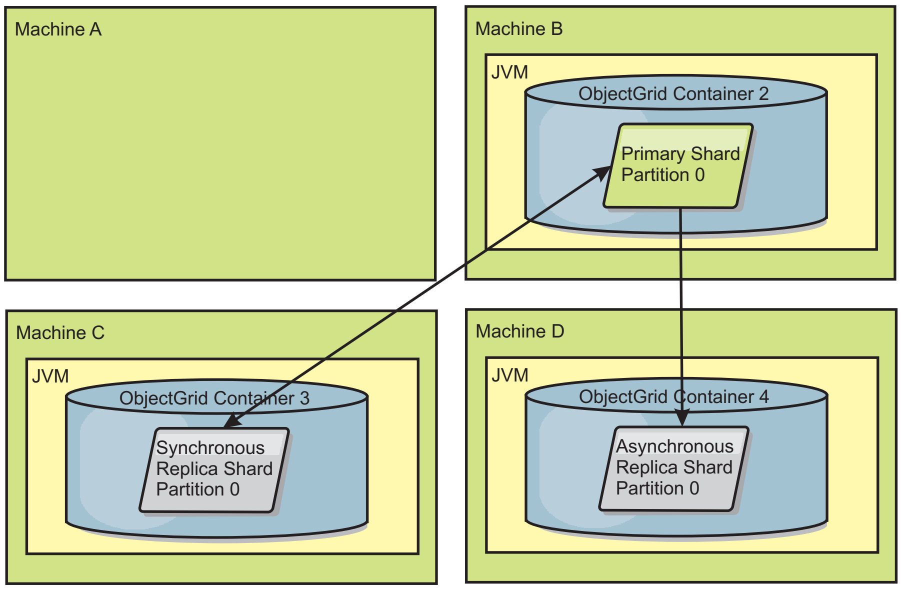


Figure 37. Machine B contains the primary shard. Depending on how automatic repair mode is set and the availability of the containers, a new synchronous replica shard might or might not be placed on a machine.

Replica shard recovery

A 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.

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.

Using 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 best way to explain this is with some examples followed by more detail.

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 JEE 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

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 associate 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.

Catalog service as the grid tie breaker

The catalog service is the tie breaker for an eXtreme Scale grid. It is essential that it acts with a single voice for the grid to execute. The catalog service runs on a fixed set of Java virtual machines replicating data from an elected primary to all the other Java virtual machines in that set. The catalog service should be distributed amongst the physical zones or data centers to lower the probability that it gets isolated from the grid and can survive anticipated failure scenarios.

The catalog service communicates with the container Java virtual machines in the grid using idempotent or recoverable operations. It does this communication using IIOP. All state changes in the catalog service are synchronously replicated amongst the current members hosting the catalog service. This replication only succeeds if the majority of the Java virtual machines accept the change. This means that if the catalog service partitions then only the majority partition can commit changes. The service primary will only send commands to the container Java virtual machines if the state change that creates that command commits. This means a minority partition cannot advance its state or issue commands to containers.

A partitioned catalog service does not stop the grid from functioning. The grid will still accept client requests and execute operations. If there is no majority catalog service partition then failures in the grid will not be recovered until the catalog service regains majority. If recovery is delayed then over time as failures occur then certain partitions will go offline until the catalog service regains majority.

The core group leader Java virtual machines report membership changes to the catalog service. If the catalog service partitions then the service will push back an updated route table for the catalog service. Such a route table from a minority partition will not include the location for a primary. The leader will need to iterate across all the possible catalog service Java virtual machines to try to locate the primary partition. It will need to do this periodically while waiting for the partition to be resolved. After it receives a route table with a primary then pending recovery actions to be directed by the majority catalog service primary.

If the core group cannot connect with a catalog service primary for a period of time then either it is physically disconnected from the rest of the grid (possibly with a minority catalog service partition) or the catalog service is stuck in minority partitions. It is impossible to tell the difference. If there is a majority catalog service partition then it may be recovering from the apparent loss of the disconnected core group. This may lead to two primaries for the same partition, the old existing primary and the new primary in the rest of the network. The majority catalog service partition has no way to 'kill' the old primaries given it's in a disconnected network state with the old primaries. When the catalog service recovers and the disconnected core group discovers the new primaries then the catalog service will notice that there are two primaries. It will instruct the previously disconnected core groups to delete all shards and then balancing will occur.

If the catalog service partitions in to two minor partitions or a single surviving minor partition then the customer will need to get involved to help with recovery. A Java Management Extensions (JMX) command is required to specify that a single minority partition is allowed to take action. You must ensure that the other minority partitions are stopped.

Zone-preferred routing

With zone-preferred routing, eXtreme Scale can direct transactions to zones based on your specifications.

WebSphere eXtreme Scale allows you to exercise a significant amount of control over where the shards of an ObjectGrid are placed. See “Using zones for replica placement” on page 99 for information on some basic scenarios and how to configure your deployment policy accordingly.

Zone-preferred routing allows eXtreme Scale clients to specify a bias for a particular zone or set of zones. So eXtreme Scale will attempt to route client transactions to preferred zones before attempting to route to any other zone.

Requirements for zone-preferred routing

There are several factors to consider before attempting zone-preferred routing. Ensure that the application is able to satisfy the requirements of your scenario.

Per-container partition placement is necessary in order to leverage zone-preferred routing. This placement strategy is a good fit for applications that are storing session data in the ObjectGrid. WebSphere eXtreme Scale's default partition placement strategy is fixed-partition. Keys are hashed at transaction commit time to determine which partition houses the key-value pair of the map when using fixed-partition placement.

Per-container placement assigns your data to a random partition at transaction commit time through the SessionHandle. You must be able to reconstruct the SessionHandle in order to retrieve your data from the ObjectGrid.

Since you can use zones to have more control over where primaries and their replicas will reside in your domain, a multi-zone deployment is advantageous when your data resides in multiple physical locations. Geographically separating primaries and replicas is a way to ensure that catastrophic loss of 1 datacenter will not impact the availability of the data.

When data is spread across a multi-zone topology, it is likely that clients are also spread across the topology. Routing clients to their local zone or datacenter has the obvious performance benefit of reduced network latency. Take advantage of this scenario when possible.

Configuring your topology for zone-preferred routing

Consider the following scenario. You have 2 data centers: Chicago and London. In order to minimize response time of clients, you want clients to read and write data to their local datacenter.

ObjectGrid primary shards must be placed in each data center so that transactions can be written locally from each location. Additionally, clients will need to be aware of zones in order to route to the local zone.

Per-container placement locates new primary shards on each container started. Replicas are placed according to zone and placement rules specified by the deployment policy. By default, a replica is placed in a different zone than its primary. Consider the following deployment policy for this scenario.

```
<?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="universe">
  <mapSet name="mapSet1" placementStrategy="PER_CONTAINER"
   numberOfPartitions="3" maxAsyncReplicas="1">
   <map ref="planet" />
  </mapSet>
 </objectgridDeployment>
</deploymentPolicy>
```

Each container that starts with the deployment policy will receive 3 new primaries. Each primary will have 1 asynchronous replica. Start each container with the appropriate zone name. Use the -zone parameter if you are launching your containers with the startOgServer script.

For a Chicago container server:

- UNIX Linux

  ```
  startOgServer.sh s1 -objectGridFile ../xml/universeGrid.xml
  -deploymentPolicyFile ../xml/universeDp.xml
  -catalogServiceEndpoints MyServer1.company.com:2809
  -zone Chicago
  ```

- Windows

  ```
  startOgServer.bat s1 -objectGridFile ../xml/universeGrid.xml
  -deploymentPolicyFile ../xml/universeDp.xml
  -catalogServiceEndpoints MyServer1.company.com:2809
  -zone Chicago
  ```

If your containers are running in WebSphere Application Server, you must create a node group and name it with the prefix “ReplicationZone”. Servers running on the nodes in such node groups will be placed into the appropriate zone. For example, servers running on a Chicago node might be in a node group named “ReplicationZoneChicago”. See the “Associating WebSphere Extended Deployment nodes with zones” in “Using zones for replica placement” on page 99.

Primaries for the Chicago zone will have replicas in the London zone. Primaries for the London zone will have replicas in the Chicago zone.

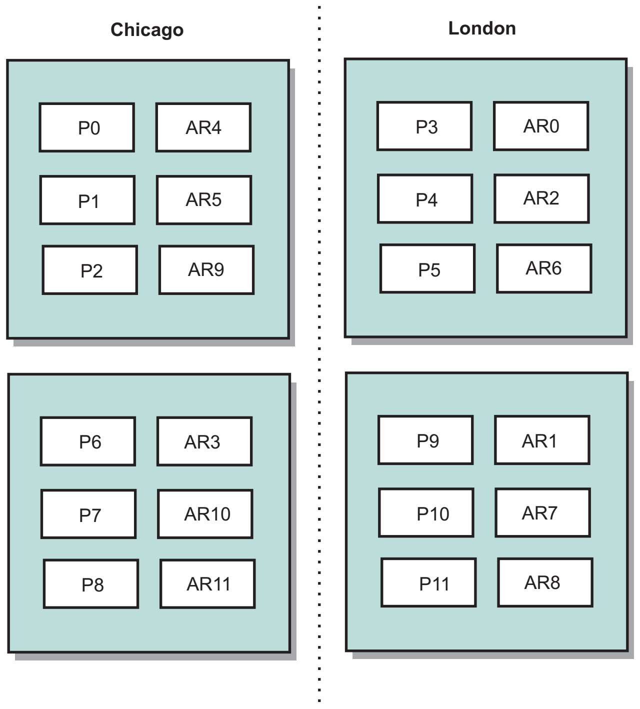


Figure 38. Primaries and replicas in zones

Set the preferred zones for the clients. This information can be provided in one of several different ways. The most straightforward way is to provide a client properties file to your client JVM. Create a file named objectGridClient.properties and ensure that it is in your class path. For more information, see the topic on the client properties file in the *Administration Guide.*

Include the preferZones property in the file. Set the property value to the appropriate zone. Clients in Chicago should have the following in the objectGridClient.properties file.

```
preferZones=Chicago
```

The property file for London clients should contain

```
preferZones=London
```

This property instructs each client to route transactions to its local zone if possible. Data that is inserted into a primary in the local zone will be asynchronously replicated to the foreign zone.

Using the SessionHandle to route to the local zone

The per-container placement strategy does not use a hash-based algorithm to determine the location of your key-value pairs in the ObjectGrid. Your ObjectGrid must leverage SessionHandles to ensure that transactions are routed to the right location when using this placement strategy. When a transaction is committed, a SessionHandle is bound to the Session if one has not already been set.

Alternatively, the SessionHandle can be bound to the Session by calling Session.getSessionHandle prior to committing the transaction. The following code snippet shows a SessionHandle being bound prior to committing the transaction.

```
Session ogSession = objectGrid.getSession();

// binding the SessionHandle
SessionHandle sessionHandle = ogSession.getSessionHandle();

ogSession.begin();
ObjectMap map = ogSession.getMap("planet");
map.insert("planet1", "mercury");

// tran is routed to partition specified by SessionHandle
ogSession.commit();
```

Assume that the code above was running on a client in your Chicago data center. Since the preferZones attribute has been set to Chicago for this client, this transaction would be routed to one of the primary partitions in the Chicago zone, partition 0, 1, 2, 6, 7, or 8.

This SessionHandle is your path back to the partition storing this committed data. The SessionHandle must be reused or reconstructed and set on the Session in order to get back to the partition containing the committed data.

```
ogSession.setSessionHandle(sessionHandle);
ogSession.begin();

// value returned will be "mercury"
String value = map.get("planet1");
ogSession.commit();
```

Since this transaction reuses the SessionHandle that was created during the insert transaction, the get transaction will be routed to the partition that holds the inserted data. This transaction will be unable to retrieve the previously inserted data if the SessionHandle has not been set.

How container and zone failures affect zone-based routing

A client with the preferZones property set will have all of its transactions routed to the specified zone or zones under normal circumstances. However, the loss of a container will result in a replica getting promoted to primary in a foreign zone. A client that was previously routing to partitions in the local zone may be forced to the foreign zone in order to retrieve previously inserted data.

Consider the following scenario. A container in the Chicago zone is lost. It previously contained primaries for partitions 0, 1, and 2. The new primaries for these partitions will be in the London zone since this zone was hosting the replicas for these partitions.

Any Chicago client that is using a SessionHandle pointing to one of the failed over partitions will now be routed to London. Chicago clients using new SessionHandles will be routed to Chicago-based primaries.

Similarly, if the entire Chicago zone is lost, all replicas in the London zone will become primaries. In this case, all Chicago clients will have their transactions routed to London.

Failover detection types

WebSphere eXtreme Scale can reliably detect failures.

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 where N is the number of missed heart beats and M is the interval at which heartbeats should be set. Directly specifying M and N is not supported, and instead, 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, such as the network interface card (NIC), which would cause the operating system to be disconnected from the network. 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 very 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 immediately sent over the connections that were being used by that process to close each connection.

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 a host machine loses power, it becomes unavailable 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.

Container failure

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 of 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 failover 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, very quickly. Typical failure detection for soft failures is less than one second. Hard failures may take up to 200 seconds to detect using the default heart beat tuning. Such failures include: physical machine crashes, network cable disconnects or operating system failures. Thus, eXtreme Scale must rely on heart beating to detect hard failures which can be configured.

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. The deployment policy defines an attribute that the catalog service uses to determine whether a replica can be placed on the same machine as a primary. In a development environment on a single machine, you might want to have two containers and replicate between them. However, in production, using a single machine is insufficient because loss of that host results in the loss of both containers. 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.

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 grid uses a peer election process for defining the primary shard instead of the catalog service algorithm that is used for the containers.

Note that the placement service and the core grouping service are one-of-N services, but the location service and administration run everywhere. 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 should restart to restore the system to the desired level of availability. If all of the processes that are hosting the catalog service fail, eXtreme Scale has a loss of critical data. This failure results in a required restart of all the containers. 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.

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

Using JMS to distribute transaction changes

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

Administration Guide

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*.

Chapter 6. Security

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.

Data 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 data 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 7. Transaction processing

WebSphere eXtreme Scale uses transactions as its mechanism for interaction with data.

Sessions and transactions

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.

Advantages of transactions

By using transactions, you can:
- Roll back changes if an exception occurs or business logic needs to undo state changes.
- 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.

Transactions

Transactions have many advantages for data storage and manipulation. You can use transactions to protect the 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.

Transaction overview

Use transactions for the following reasons:
- To protect a thread from concurrent changes.
- To apply multiple changes as an atomic unit at commit time.
- To implement a life cycle for locks on changes.
- To act as the unit of replication.

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.

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.

CopyMode attribute

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.

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.

Locking mode and transactions

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" />
```

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. See the *Programming Guide* for more information about eXtreme Scale and external transaction interaction. If you are using WebSphere Application Server, you can use the WebSphereTranscationCallback plug-in. See the *Programming Guide* for more information about the plug-ins that are available with WebSphere eXtreme Scale.

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.

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 120.

For more information, see the topic on handling locks in the *Programming Guide* or map entry locking in the *Administration Guide*.

Chapter 8. Tutorials

You can use tutorials to get started with particular WebSphere eXtreme Scale functions.

Tutorials, examples, and samples

Several WebSphere eXtreme Scale tutorials, examples and samples are available.

Tutorials

The following tutorials are currently available.
- ObjectMap tutorial
- "Entity manager tutorial: Overview"
- "Java SE security tutorial - Main page" on page 139
- ObjectQuery tutorial

Examples

The topics below illustrate key WebSphere eXtreme Scale features.
- See the Data Grid API example in the *Programming Guide*
- See the details on configuring a local eXtreme Scale configuration in the *Administration Guide*

Samples

Samples to illustrate how to use ObjectGrid APIs in various environments are shipped with the WebSphere eXtreme Scale product.

Articles with tutorials and examples

Table 10. 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

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 124 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 EntityManager 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 39. 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" on page 133
   - 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 134
   - How to create an index that the query can use
3. "ObjectQuery tutorial - step 3" on page 135
   - 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 137
   - 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 133 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 will automatically use the index when there. This 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 134 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 135 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 - Main page

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"
   - 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 143
   - 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 149
   - Use of JAAS authorization policy
4. "Java SE security tutorial - Step 4" on page 152
   - 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 139.

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:

```
// 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.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, launch 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
   ```

   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 Scaleauthorization mechanisms.

Before you begin

Be sure to have completed “Java SE security tutorial - Step 2” on page 143 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 `bin` 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 149 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. **ConfigureObjectGrid 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 149 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 143 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.

Chapter 9. Glossary

This glossary includes terms and definitions for WebSphere eXtreme Scale.

The following cross-references are used in this glossary:

1. See refers the reader from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
2. See also refers the reader to a related or contrasting term.

To view glossaries for other IBM products, go to www.ibm.com/software/globalization/terminology.

administrator. A person responsible for administrative tasks such as access authorization and content management. Administrators can also grant levels of authority to users.

agent. A program that performs an action on behalf of a user or other program without user intervention or on a regular schedule, and reports the results back to the user or program.

APAR. See authorized program analysis report.

API. See application programming interface.

application. One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application programming interface (API). An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

application server. A server program in a distributed network that provides the execution environment for an application program.

asynchronous. Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals.

asynchronous messaging. A method of communication between programs in which a program places a message on a message queue, then proceeds with its own processing without waiting for a reply to its message.

asynchronous replica. A shard that receives updates after the transaction commits. This method is faster than a synchronous replica, but introduces the possibility of data loss because the asynchronous replica can be several transactions behind the primary shard.

authenticated user. A portal user who has logged in to the portal with a valid account (user ID and password). Authenticated users have access to all public places.

authentication. A security service that provides proof that a user of a computer system is genuinely who that person claims to be. Common mechanisms for implementing this service are passwords and digital signatures. Authentication is distinct from authorization; authentication is not concerned with granting or denying access to system resources.

authentication alias. An alias that authorizes access to resource adapters and data sources. An authentication alias contains authentication data, including a user ID and password.

authorization. The process of granting a user, system, or process either complete or restricted access to an object, resource, or function.

authorization policy. A policy whose policy target is a business service and whose contract contains one or more assertions that grant permission to run a channel action.

authorization table. A table that contains the role to user or group mapping information that identifies the permitted access of a client to a particular resource.

authorized program analysis report (APAR). A request for correction of a defect in a supported release of an IBM-supplied program.

autodiscovery. The discovery of service artifacts in a file system, external registry, or another source.

autonomic manager. A set of software or hardware components, configured by policies, which manage the behavior of other software or hardware components as a human might manage them. An autonomic manager includes a control loop that consists of monitor, analyze, plan, and execute components.

availability.

1. The condition allowing users to access and use their applications and data.

2. The time periods during which a resource is accessible. For example, a contractor might have an availability of 9 AM to 5 PM every weekday, and 9 AM to 3 PM on Saturdays.

bean. A definition or instance of a JavaBeans component. See also JavaBeans, enterprise bean.

bean class. In Enterprise JavaBeans (EJB) programming, a Java class that implements a javax.ejb.EntityBean class or javax.ejb.SessionBean class.

Bean Scripting Framework. An architecture for incorporating scripting language functions to Java applications.

bean-managed messaging. A function of asynchronous messaging that gives an enterprise bean complete control over the messaging infrastructure.

bean-managed persistence (BMP). The mechanism whereby data transfer between an entity bean's variables and a resource manager is managed by the entity bean. (Sun)

bean-managed transaction (BMT). The capability of the session bean, servlet, or application client component to manage its own transactions directly, instead of through a container.

binary format. Representation of a decimal value in which each field must be 2 or 4 bytes long. The sign (+ or -) is in the far left bit of the field, and the number value is in the remaining bits of the field. Positive numbers have a 0 in the sign bit and are in true form. Negative numbers have a 1 in the sign bit and are in twos complement form.

BMP. See bean-managed persistence.

BMT. See bean-managed transaction.

bootstrap. A small program that loads larger programs during system initialization.

bootstrapping. The process by which an initial reference of the naming service is obtained. The bootstrap setting and the host name form the initial context for Java Naming and Directory Interface (JNDI) references.

bottleneck. A place in the system where contention for a resource is affecting performance.

bottom-up development. In Web services, the process of developing a service from an existing artifact such as a Java bean or enterprise bean rather than a Web Services Description Language (WSDL) file.

breakpoint. A marked point in a process or programmatic flow that causes that flow to pause when the point is reached, typically to allow debugging or monitoring.

build definition file. An XML file that identifies components and characteristics for a customized installation package (CIP).

build path. The path that is used during compilation of Java source code, in order to find referenced classes that reside in other projects.

build plan. An XML file that defines the processing necessary to build generation outputs and that specifies the machine where processing takes place.

build time data. Objects that are not used by the translator, such as EDI standards, record oriented data document types, and maps.

bytecode. Machine-independent code generated by the Java compiler and executed by the Java interpreter. (Sun)

cache instance resource. A location where any Java Platform, Enterprise Edition (Java EE) application can store, distribute, and share data.

cache replication. The sharing of cache IDs, cache entries, and cache invalidations with other servers in the same replication domain.

catalog. A container that, depending on the container type, holds processes, data, resources, organizations, or reports in the project tree.

catalog service. A service that controls placement of shards and discovers and monitors the health of containers.

category. A container used in a structure diagram to group elements based on a shared attribute or quality.

cell.

1. A group of managed processes that are federated to the same deployment manager and can include high-availability core groups.

2. One or more processes that each host runtime components. Each has one or more named core groups.

cell-scoped binding. A binding scope where the binding is not specific to, and not associated with any node or server. This type of name binding is created under the persistent root context of a cell.

chassis. The metal frame in which various electronic components are mounted.

child node. A node within the scope of another node.

CIP. See customized installation package.

class. In object-oriented design or programming, a model or template that can be used to create objects with a common definition and common properties, operations, and behavior. An object is an instance of a class.

class file. A compiled Java source file.

class hierarchy. The relationships between classes that share a single inheritance.

class loader. Part of the Java virtual machine (JVM) that is responsible for finding and loading class files. A class loader affects the packaging of applications and the runtime behavior of packaged applications deployed on application servers.

class path. A list of directories and JAR files that contain resource files or Java classes that a program can load dynamically at run time.

classifier. A specialized attribute used for grouping and color-coding process elements.

client. A software program or computer that requests services from a server. See also host.

client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

client/server. Pertaining to the model of interaction in distributed data processing in which a program on one computer sends a request to a program on another computer and awaits a response. The requesting program is called a client; the answering program is called a server.

Cloudscape. An embeddable, all Java, object-relational database management system (ORDBMS).

cluster. A group of application servers that collaborate for the purposes of workload balancing and failover.

coarse-grained. Pertaining to viewing a group of objects from an abstract or high level.

coherent cache. Cache that maintains integrity so that all clients see the same data.

collection certificate store. A collection of intermediate certificates or certificate revocation lists (CRL) that are used by a certificate path to build up a certificate chain for validation.

comma delimited file. A file whose records contain fields that are separated by a comma.

command bean. A proxy that can invoke a single operation using an execute() method.

command line. The blank line on a display where commands, option numbers, or selections can be entered.

compilation unit. A portion of a computer program sufficiently complete to be compiled correctly.

compile time. The time period during which a computer program is being compiled into an executable program.

component.

1. A reusable object or program that performs a specific function and works with other components and applications.

2. In Eclipse, one or more plug-ins that work together to deliver a discrete set of functions.

component element. An entity in a component where a breakpoint can be set, such as an activity or Java snippet in a business process, or a mediation primitive or node in a mediation flow.

component instance. A running component that can be running in parallel with other instances of the same component.

component test. An automated test of one or more components of an enterprise application, which may include Java classes, EJB beans, or Web services.

container server. A server instance that can host multiple shards. One Java virtual machine (JVM) can host multiple container servers.

converter. In Enterprise JavaBeans (EJB) programming, a class that translates a database representation to an object type and back.

create method. In enterprise beans, a method defined in the home interface and invoked by a client to create an enterprise bean. (Sun)

credential. In the Java Authentication and Authorization Service (JAAS) framework, a subject class that owns security-related attributes. These attributes can contain information used to authenticate the subject to new services.

customized installation package (CIP). A customized installation image that can include one or more maintenance packages, a configuration archive file from a stand-alone server profile, one or more enterprise archive files, scripts, and other files that help customize the resulting installation.

daemon. A program that runs unattended to perform continuous or periodic functions, such as network control.

dashboard. A Web page that can contain one or more viewers that graphically represent business data.

data grid. A system for accessing terabytes or petrabytes of data.

DB2. A family of IBM licensed programs for relational database management.

deadlock. A condition in which two independent threads of control are blocked, each waiting for the other to take some action. Deadlock often arises from adding synchronization mechanisms to avoid race conditions.

demilitarized zone (DMZ). A configuration that includes multiple firewalls to add layers of protection between a corporate intranet and a public network, such as the Internet.

deploy. To place files or install software into an operational environment. In Java Platform, Enterprise Edition (Java EE), this involves creating a deployment descriptor suitable to the type of application that is being deployed.

deploy phase. See deployment phase.

deployment code. Additional code that enables bean implementation code written by an application developer to work in a particular EJB runtime environment. Deployment code can be generated by tools that the application server vendor supplies.

deployment descriptor. An XML file that describes how to deploy a module or application by specifying configuration and container options. For example, an EJB deployment descriptor passes information to an EJB container about how to manage and control an enterprise bean.

deployment directory. The directory where the published server configuration and Web application are located on the machine where the application server is installed.

deployment environment. A collection of configured clusters, servers, and middleware that collaborate to provide an environment to host software modules. For example, a deployment environment might include a host for message destinations, a processor or sorter of business events, and administrative programs.

deployment manager. A server that manages operations for a logical group or cell of other servers.

deployment phase. A phase that includes a combination of creating the hosting environment for your applications and the deployment of those applications. This includes resolving the application's resource dependencies, operational conditions, capacity requirements, and integrity and access constraints.

deployment policy. An optional way to configure an eXtreme Scale environment based on various items, including: number of systems, servers, partitions, replicas (including type of replica), and heap sizes for each server.

deployment topology. The configuration of servers and clusters in a deployment environment and the physical and logical relationships among them.

deprecated. Pertaining to an entity, such as a programming element or feature, that is supported but no longer recommended and that might become obsolete.

derivation. In object-oriented programming, the refinement or extension of one class from another.

deserialization. A method for converting a serialized variable into object data.

destination. An exit point that is used to deliver documents to a back-end system or a trading partner.

digital certificate. An electronic document used to identify an individual, a system, a server, a company, or some other entity, and to associate a public key with the entity. A digital certificate is issued by a certification authority and is digitally signed by that authority.

dirty read. A read request that does not involve any locking mechanism. This means that data can be read that might later be rolled back resulting in an inconsistency between what was read and what is in the database.

distributed eXtreme Scale. A usage pattern for interacting with eXtreme Scale when servers and clients exist on multiple processes.

DMZ. See demilitarized zone.

DNS. See Domain Name System.

do-while loop. A loop that repeats the same sequence of activities as long as some condition is satisfied. Unlike a while loop, a do-while loop tests its condition at the end of the loop. This means that its sequence of activities always runs at least once.

document type definition (DTD). The rules that specify the structure for a particular class of SGML or XML documents. The DTD defines the structure with elements, attributes, and notations, and it establishes constraints for how each element, attribute, and notation can be used within the particular class of documents.

domain. An object, icon, or container that contains other objects representing the resources of a domain. The domain object can be used to manage those resources.

Domain Name System (DNS). The distributed database system that maps domain names to IP addresses.

downstream. Pertaining to the direction of the flow, which is from the first node in the process (upstream) toward the last node in the process (downstream).

drop-down. See pull-down.

DTD. See document type definition.

DTD document definition. A description or layout of an XML document based on an XML DTD.

dynamic cache. A consolidation of several caching activities, including servlets, Web services, and WebSphere commands into one service where these activities share configuration parameters and work together to improve performance.

dynamic cluster. A server cluster that uses weights to balance the workloads of its cluster members dynamically, based on performance information collected from cluster members.

EAR. See enterprise archive.

EAR project. See enterprise application project.

Eclipse. An open-source initiative that provides ISVs and other tool developers with a standard platform for developing plug-compatible application development tools.

edition. A successive deployment generation of a particular set of versioned artifacts.

editor area. In Eclipse and Eclipse-based products, the area in the workbench window where files are opened for editing.

EJB. See Enterprise JavaBeans.

EJB container. A container that implements the EJB component contract of the Java EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. (Sun)

EJB context. In enterprise beans, an object that allows an enterprise bean to invoke services provided by the container and to obtain information about the caller of a client-invoked method. (Sun)

EJB factory. An access bean that simplifies the creating or finding of an enterprise bean instance.

EJB home object. In Enterprise JavaBeans (EJB) programming, an object that provides the life cycle operations (create, remove, find) for an enterprise bean. (Sun)

EJB inheritance. A form of inheritance in which an enterprise bean inherits properties, methods, and method-level control descriptor attributes from another enterprise bean that resides in the same group.

EJB JAR file. A Java archive that contains an EJB module. (Sun)

EJB module. A software unit that consists of one or more enterprise beans and an EJB deployment descriptor. (Sun)

EJB object. In enterprise beans, an object whose class implements the enterprise bean remote interface (Sun).

EJB project. A project that contains the resources needed for EJB applications, including enterprise beans; home, local, and remote interfaces; JSP files; servlets; and deployment descriptors.

EJB query. In EJB query language, a string that contains an optional SELECT clause specifying the EJB objects to return, a FROM clause that names the bean collections, an optional WHERE clause that contains search predicates over the collections, an optional ORDER BY clause that specifies the ordering of the result collection, and input parameters that correspond to the arguments of the finder method.

EJB reference. A logical name used by an application to locate the home interface of an enterprise bean in the target operational environment.

EJB server. Software that provides services to an EJB container. An EJB server may host one or more EJB containers. (Sun)

embedded server. A catalog service or container server that resides in an existing process and is started and stopped within the process.

endpoint.

1. A JCA application or other client consumer of an event from the enterprise information system.

2. The system that is the origin or destination of a session.

endpoint listener. The point or address at which incoming messages for a Web service are received by a service integration bus.

enterprise application project (EAR project). A structure and hierarchy of folders and files that contain a deployment descriptor and IBM extension document as well as files that are common to all Java EE modules that are defined in the deployment descriptor.

enterprise archive (EAR). A specialized type of JAR file, defined by the Java EE standard, used to deploy Java EE applications to Java EE application servers. An EAR file contains EJB components, a deployment descriptor, and Web archive (WAR) files for individual Web applications. See also Web archive.

enterprise bean. A component that implements a business task or business entity and resides in an EJB container. Entity beans, session beans, and message-driven beans are all enterprise beans. (Sun) See also bean.

Enterprise JavaBeans (EJB). A component architecture defined by Sun Microsystems for the development and deployment of object-oriented, distributed, enterprise-level applications (Java EE).

enterprise service bus (ESB). A flexible connectivity infrastructure for integrating applications and services; it offers a flexible and manageable approach to service-oriented architecture implementation.

entity.

1. A simple Java class that represents a row in a database table or entry in a map.

2. In markup languages such as XML, a collection of characters that can be referenced as a unit, for example to incorporate often-repeated text or special characters within a document.

entity bean. In EJB programming, an enterprise bean that represents persistent data maintained in a database. Each entity bean carries its own identity. (Sun)

entry breakpoint. A breakpoint set on a component element that is hit before the component element is invoked.

environment. A named collection of logical and physical resources used to support the performance of a function.

environment variable. A variable that specifies how an operating system or another program runs, or the devices that the operating system recognizes.

error. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

error log stream. A continuous flow of error information that is transmitted using a predefined format.

ESB. See enterprise service bus.

event.

1. A change to a state, such as the completion or failure of an operation, business process, or human task, that can trigger a subsequent action, such as persisting the event data to a data repository or invoking another business process.

2. A change to data in an enterprise information system (EIS) that is processed by the adapter and used to deliver business objects from the EIS to the endpoints (applications) that need to be notified of the change.

evictor. A component that controls the membership of entries in each BackingMap instance. Sparse caches can use evictors to automatically remove data from the cache without affecting the database.

exception. A condition or event that cannot be handled by a normal process.

exception handler. A set of routines that responds to an abnormal condition. An exception handler is able to interrupt and to resume the normal running of processes.

exclusive lock. A lock that prevents concurrently executing application processes from accessing database data. See also shared lock.

execution trace. A chain of events that is recorded and displayed in a hierarchal format on the Events page of the integration test client.

export. An exposed interface from a Service Component Architecture (SCA) module that offers a business service to the outside world. An export has a binding that defines how the service can be accessed by service requesters, for example, as a Web service.

export file.

1. A file created during the development process for inbound operations that contains the configuration settings for inbound processing.

2. The file containing data that has been exported.

expression. An SQL or XQuery operand or a collection of SQL or XQuery operators and operands that yields a single value.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that is based on Standard Generalized Markup Language (SGML).

eXtreme Scale grid. A pattern that is used to interact with eXtreme Scale when all of the data and clients are in one process.

factory. In object-oriented programming, a class that is used to create instances of another class. A factory is used to isolate the creation of objects of a particular class into one place so that new functions can be provided without widespread code changes.

failover. An automatic operation that switches to a redundant or standby system in the event of a software, hardware, or network interruption.

fine-grained. Pertaining to viewing an individual object in detail.

fire. In object-oriented programming, to cause a state transition.

firewall. A network configuration, typically both hardware and software, that prevents unauthorized traffic into and out of a secure network.

fix pack. A cumulative collection of fixes that is made available between scheduled refresh packs, manufacturing refreshes, or releases. It is intended to allow customers to move to a specific maintenance level. See also interim fix.

folder. A container used to organize objects.

for loop. A loop that repeats the same sequence of activities a specified number of times.

fork. A process element that makes copies of its input and forwards them by several processing paths in parallel.

garbage collection. A routine that searches memory to reclaim space from program segments or inactive data.

General Inter-ORB Protocol (GIOP). A protocol that Common Object Request Broker Architecture (CORBA) uses to define the format of messages.

generic object. An object that is used in API calls and XPATH expressions to refer to concepts, custom entities, or collections. For example, the XPATH expression /WSRR/GenericObject will retrieve all concepts from WebSphere Service Registry and Repository.

getter method. A method whose purpose is to get the value of an instance or class variable. This allows another object to find out the value of one of its variables.

GIOP. See General Inter-ORB Protocol.

global.

1. Pertaining to an element that is available to any process in the workspace. A global element appears in the project tree and can be used in multiple processes. Tasks, processes, repositories, and services can be either global (referenced by any process in the project) or local (specific to a single process).

2. Pertaining to information available to more than one program or subroutine.

global attribute. In XML, an attribute that is declared as a child of the schema element rather than as part of a complex type definition. Global attributes can be referenced in one or more content models using the ref attribute.

global element. In XML, an element that is declared as a child of the schema element rather than as part of a complex type definition. Global elements can be referenced in one or more content models using the ref attribute.

global instance identifier. A globally unique identifier that is generated either by the application or by the emitter and is used as a primary key for event identification.

global security. Pertains to all applications running in the environment and determines whether security is used, the type of registry used for authentication, and other values, many of which act as defaults.

global transaction. A recoverable unit of work performed by one or more resource managers in a distributed transaction environment and coordinated by an external transaction manager.

global variable. A variable that is used to hold and manipulate values assigned to it during translation and that is shared across maps and across document translations. One of the three types of variables supported by the Data Interchange Services mapping command language.

group.

1. A collection of users who can share access authorities for protected resources.

2. A set of related documents within an interchange. An interchange can contain zero to many groups.

3. In places, two or more people who are grouped for membership in a place.

HA. See high availability.

HA group. A collection of one or more members used to provide high availability for a process.

HA policy. A set of rules that is defined for an HA group that dictate whether zero (0), or more members are activated. The policy is associated with a specific HA group by matching the policy match criteria with the group name.

health. The general condition or state of the database environment.

heartbeat. A signal that one entity sends to another to convey that it is still active.

high availability (HA). Pertaining to a clustered system that is reconfigured when node or daemon failures occur, so that workloads can be redistributed to the remaining nodes in the cluster.

high availability manager. A framework within which core group membership is determined and status is communicated between core group members.

host.

1. A computer that is connected to a network and that provides an access point to that network. The host can be a client, a server, or both a client and server simultaneously.

2. In performance profiling, a machine that owns processes that are being profiled. See also server.

host name.

1. In Internet communication, the name given to a computer. The host name might be a fully qualified domain name such as mycomputer.city.company.com, or it might be a specific subname such as mycomputer.

2. The network name for a network adapter on a physical machine in which the node is installed.

host system. An enterprise mainframe computer system that hosts 3270 applications. In the 3270 terminal service development tools, the developer uses the 3270 terminal service recorder to connect to the host system.

HTTP over SSL (HTTPS). A Web protocol for secure transactions that encrypts and decrypts user page requests and pages returned by the Web server.

HTTPS.

1. See HTTP over SSL.

2. See Hypertext Transfer Protocol Secure.

Hypertext Transfer Protocol Secure (HTTPS). An Internet protocol that is used by Web servers and Web browsers to transfer and display hypermedia documents securely across the Internet.

IDE. See integrated development environment.

if-then rule. A rule in which the action (then part) is performed only when the condition (if part) is true.

IIOP. See Internet Inter-ORB Protocol.

import.

1. A development artifact that imports a service that is external to a module.

2. The point at which an SCA module accesses an external service, (a service outside the SCA module) as if it was local. An import defines interactions between the SCA module and the service provider. An import has a binding and one or more interfaces.

index. A set of pointers that are logically ordered by the values of a key. Indexes provide quick access to data and can enforce uniqueness of the key values for the rows in the table.

information center. A collection of information that provides support for users of one or more products, can be launched separately from the product, and includes a list of topics for navigation and a search engine.

inheritance. An object-oriented programming technique in which existing classes are used as a basis for creating other classes. Through inheritance, more specific elements incorporate the structure and behavior of more general elements.

installation package. An installable unit of a software product. Software product packages are separately installable units that can operate independently from other packages of that software product.

installation target. The system on which selected installation packages are installed.

instance. A specific occurrence of an object that belongs to a class.

instantiate. To represent an abstraction with a concrete instance.

integrated development environment (IDE). A set of software development tools, such as source editors, compilers, and debuggers, that are accessible from a single user interface.

interface. A collection of operations that are used to specify a service of a class or a component.

interim fix. A certified fix that is generally available to all customers between regularly scheduled fix packs, refresh packs, or releases. See also fix pack.

Internet Inter-ORB Protocol (IIOP). A protocol used for communication between Common Object Request Broker Architecture (CORBA) object request brokers.

Internet Protocol (IP). A protocol that routes data through a network or interconnected networks. This protocol acts as an intermediary between the higher protocol layers and the physical network.

invocation. The activation of a program or procedure.

IP. See Internet Protocol.

IP sprayer. A device that is located between inbound requests from the users and the application server nodes that reroutes requests across nodes.

iteration. See loop.

iterator. A class or construct that is used to step through a collection of objects one at a time.

JAAS. See Java Authentication and Authorization Service.

JAF. See JavaBeans Activation Framework.

JAR file. A Java archive file. See also Web archive, enterprise archive.

Java. An object-oriented programming language for portable interpretive code that supports interaction among remote objects. Java was developed and specified by Sun Microsystems, Incorporated.

Java API for XML (JAX). A set of Java-based APIs for handling various operations involving data defined through Extensible Markup Language (XML).

Java archive. A compressed file format for storing all of the resources that are required to install and run a Java program in a single file. See also Web archive, enterprise archive.

Java Authentication and Authorization Service (JAAS). In Java EE technology, a standard API for performing security-based operations. Through JAAS, services can authenticate and authorize users while enabling the applications to remain independent from underlying technologies.

Java class. A class that is written in the Java language.

Java Command Language. A scripting language for the Java environment that is used to create Web content and to control Java applications.

Java Connector security. An architecture designed to extend the end-to-end security model for Java EE-based applications to include enterprise information systems (EIS).

Java Database Connectivity (JDBC). An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call level interface for SQL-based and XQuery-based database access.

Java EE. See Java Platform, Enterprise Edition.

Java EE application. Any deployable unit of Java EE functionality. This unit can be a single module or a group of modules packaged into an enterprise archive (EAR) file with a Java EE application deployment descriptor. (Sun)

Java EE Connector Architecture (JCA). A standard architecture for connecting the Java EE platform to heterogeneous enterprise information systems (EIS).

Java EE server. A runtime environment that provides EJB or Web containers.

Java file. An editable source file (with .java extension) that can be compiled into bytecode (a .class file).

Java Management Extensions (JMX). A means of doing management of and through Java technology. JMX is a universal, open extension of the Java programming language for management that can be deployed across all industries, wherever management is needed.

Java Message Service (JMS). An application programming interface that provides Java language functions for handling messages.

Java Naming and Directory Interface (JNDI). An extension to the Java platform that provides a standard interface for heterogeneous naming and directory services.

Java platform. A collective term for the Java language for writing programs; a set of APIs, class libraries, and other programs used in developing, compiling, and error-checking programs; and a Java virtual machine which loads and runs the class files. (Sun)

Java Platform, Enterprise Edition (Java EE). An environment for developing and deploying enterprise applications, defined by Sun Microsystems Inc. The Java EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Sun)

Java Platform, Standard Edition (Java SE). The core Java technology platform. (Sun)

Java project. In Eclipse, a project that contains compilable Java source code and is a container for source folders or packages.

Java runtime environment. A subset of a Java developer kit that contains the core executable programs and files that constitute the standard Java platform. The JRE includes the Java virtual machine (JVM), core classes, and supporting files.

Java SE. See Java Platform, Standard Edition.

Java SE Development Kit (JDK). The name of the software development kit that Sun Microsystems provides for the Java platform.

Java Secure Socket Extension (JSSE). A Java package that enables secure Internet communications. It implements a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TSL) protocols and supports data encryption, server authentication, message integrity, and optionally client authentication.

Java Specification Request (JSR). A formally proposed specification for the Java platform.

Java virtual machine (JVM). A software implementation of a processor that runs compiled Java code (applets and applications).

Java virtual machine Profiler Interface (JVMPI). A profiling tool that supports the collection of information, such as data about garbage collection and the Java virtual machine (JVM) API that runs the application server.

JavaBeans. As defined for Java by Sun Microsystems, a portable, platform-independent, reusable component model. See also bean.

JavaBeans Activation Framework (JAF). A standard extension to the Java platform that determines arbitrary data types and available operations and can instantiate a bean to run pertinent services.

Javadoc.

1. A tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields. (Sun)

2. Pertaining to the tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields.

JavaMail API. A platform and protocol-independent framework for building Java-based mail client applications.

JavaScript. A Web scripting language that is used in both browsers and Web servers. (Sun)

JavaScript Object Notation. A lightweight data-interchange format that is based on the object-literal notation of JavaScript. JSON is programming-language neutral but uses conventions from languages that include C, C++, C#, Java, JavaScript, Perl, Python.

JavaServer Pages (JSP). A server-side scripting technology that enables Java code to be dynamically embedded within Web pages (HTML files) and run when the page is served, in order to return dynamic content to a client.

JAX. See Java API for XML.

JCA. See Java EE Connector Architecture.

JDBC. See Java Database Connectivity.

JDK. See Java SE Development Kit.

JMS. See Java Message Service.

JMS data binding. A data binding that provides a mapping between the format used by an external JMS message and the Service Data Object (SDO) representation used by a Service Component Architecture (SCA) module.

JMX. See Java Management Extensions.

JNDI. See Java Naming and Directory Interface.

join.

1. A process element that recombines and synchronizes parallel processing paths after a decision or fork. A join waits for input to arrive at each of its incoming branches before permitting the process to continue.

2. An SQL relational operation in which data can be retrieved from two tables, typically based on a join condition specifying join columns.

3. The configuration on an incoming link that determines the behavior of the link.

JSP. See JavaServer Pages.

JSP file. A scripted HTML file that has a .jsp extension and allows for the inclusion of dynamic content in Web pages. A JSP file can be directly requested as a URL, called by a servlet, or called from within an HTML page.

JSP page. A text-based document using fixed template data and JSP elements that describes how to process a request to create a response. (Sun)

JSR. See Java Specification Request.

JSSE. See Java Secure Socket Extension.

JVM. See Java virtual machine.

JVMPI. See Java virtual machine Profiler Interface.

Jython. An implementation of the Python programming language that is integrated with the Java platform.

key.

1. A cryptographic mathematical value that is used to digitally sign, verify, encrypt, or decrypt a message.

2. Information that characterizes and uniquely identifies the real-world entity that is being tracked by a monitoring context.

keyword. One of the predefined words of a programming language, artificial language, application, or command.

LDAP. See Lightweight Directory Access Protocol.

LDAP directory. A type of repository that stores information on people, organizations, and other resources and that is accessed using the LDAP protocol. The entries in the repository are organized into a hierarchical structure, and in some cases the hierarchical structure reflects the structure or geography of an organization.

library.

1. A collection of model elements, including business items, processes, tasks, resources, and organizations.

2. A project that is used for the development, version management, and organization of shared resources. Only a subset of the artifact types can be created and stored in a library, such as business objects and interfaces.

life cycle. One complete pass through the four phases of software development: inception, elaboration, construction and transition.

Lightweight Directory Access Protocol (LDAP). An open protocol that uses TCP/IP to provide access to directories that support an X.500 model and that does not incur the resource requirements of the more complex X.500 Directory Access Protocol (DAP). For example, LDAP can be used to locate people, organizations, and other resources in an Internet or intranet directory.

Lightweight Third Party Authentication (LTPA). A protocol that uses cryptography to support security in a distributed environment.

listener. A program that detects incoming requests and starts the associated channel.

listener port. An object that defines the association between a connection factory, a destination, and a deployed message-driven bean. Listener ports simplify the administration of the associations between these resources.

load balancing. The monitoring of application servers and management of the workload on servers. If one server exceeds its workload, requests are forwarded to another server with more capacity.

loader. A component that reads data from and writes data to a persistent store.

local.

1. Pertaining to a device, file, or system that is accessed directly from a user system, without the use of a communication line.

2. Pertaining to an element that is available only in its own process.

local database. A database that is located on the workstation in use.

lock. A means of preventing uncommitted changes made by one application process from being perceived by another application process and for preventing one application process from updating data that is being accessed by another process. A lock ensures the integrity of data by preventing concurrent users from accessing inconsistent data.

logging. The recording of data about specific events on the system, such as errors.

long name. The property that specifies the logical name for the server on the z/OS® platform.

loop. A sequence of instructions performed repeatedly.

LTPA. See Lightweight Third Party Authentication.

maintenance mode. A state of a node or server that an administrator can use to diagnose, maintain, or tune the node or server without disrupting incoming traffic in a production environment.

Managed Bean (MBean). In the Java Management Extensions (JMX) specification, the Java objects that implement resources and their instrumentation.

map.

1. A data structure that maps keys to values.

2. A file that defines the transformation between sources and targets.

3. In the EJB development environment, the specification of how the container-managed persistent fields of an enterprise bean correspond to columns in a relational database table or other persistent storage.

MBean. See Managed Bean.

MBean provider. A library containing an implementation of a Java Management Extensions (JMX) MBean and its MBean Extensible Markup Language (XML) descriptor file.

memory leak. The effect of a program that maintains references to objects that are no longer required and therefore need to be reclaimed.

method. In object-oriented programming, an operation that an object can perform. An object can have many methods.

metric. A holder for information, typically a business performance measurement, in a monitoring context.

namespace. A logical container in which all the names are unique. The unique identifier for an artifact is composed of the namespace and the local name of the artifact.

node.

1. A logical grouping of managed servers.

2. Any item on a tree control, including a simple element, compound element, mapping command, comment, or group node.

3. In XML, the smallest unit of valid, complete structure in a document.

4. The fundamental shapes that make up a diagram.

node agent. An administrative agent that manages all application servers on a node and represents the node in the management cell.

object. In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data.

Object Request Broker (ORB). In object-oriented programming, software that serves as an intermediary by transparently enabling objects to exchange requests and responses.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished but instead on what data objects comprise the problem and how they are manipulated.

ObjectGrid. A grid-enabled, memory database for applications that are written in Java. ObjectGrid can be used as an in-memory database or to distribute data across a network.

ODBC. See Open Database Connectivity.

Open Database Connectivity (ODBC). A standard application programming interface (API) for accessing data in both relational and nonrelational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface.

open source. Pertaining to software whose source code is publicly available for use or modification. Open source software is typically developed as a public collaboration and made freely available, although its use and redistribution might be subject to licensing restrictions. Linux is a well known example of open source software.

operation. An implementation of functions or queries that an object might be called to perform.

ORB. See Object Request Broker.

organization. An entity where people cooperate to accomplish specified objectives, such as an enterprise, a company, or a factory.

package.

1. In Java programming, a group of types. Packages are declared with the package keyword. (Sun)

2. The wrapper around the document content that defines the format used to transmit a document over the Internet, for example, RNIF, AS1, and AS2.

3. To assemble components into modules and modules into enterprise applications.

partitioning facility. A programming framework and a system management infrastructure that supports the concept of partitioning for enterprise beans, HTTP traffic, and database access.

Performance Monitoring Infrastructure (PMI). A set of packages and libraries assigned to gather, deliver, process, and display performance data.

permission. Authorization to perform activities, such as reading and writing local files, creating network connections, and loading native code.

persist. To be maintained across session boundaries, typically in nonvolatile storage such as a database system or a directory.

persistence.

1. A characteristic of data that is maintained across session boundaries, or of an object that continues to exist after the execution of the program or process that created it, typically in nonvolatile storage such as a database system.

2. In Java EE, the protocol for transferring the state of an entity bean between its instance variables and an underlying database. (Sun)

persistent data store. A nonvolatile storage for event data, such as a database system, that is maintained across session boundaries and that continues to exist after the execution of the program or process that created it.

pessimistic locking. A locking strategy whereby a lock is held between the time that a row is selected and the time that a searched update or delete operation is attempted on that row.

plug-in. A separately installable software module that adds function to an existing program, application, or interface.

PMI. See Performance Monitoring Infrastructure.

point-to-point. Pertaining to a style of messaging application in which the sending application knows the destination of the message.

policy. A set of considerations that influence the behavior of a managed resource or a user.

port. As defined in a Web Services Description Language (WSDL) document, a single endpoint that is defined as a combination of a binding and a network address.

port number. In Internet communications, the identifier for a logical connector between an application entity and the transport service.

primary key.

1. An object that uniquely identifies an entity bean of a particular type.

2. In a relational database, a key that uniquely identifies one row of a database table.

primitive type. In Java, a category of data type that describes a variable that contains a single value of the appropriate size and format for its type: a number, a character, or a Boolean value. Examples of primitive types include byte, short, int, long, float, double, char, boolean.

process.

1. A progressively continuing procedure consisting of a series of controlled activities that are systematically directed toward a particular result or end.

2. The sequence of documents or messages to be exchanged between the Community Managers and participants to run a business transaction.

profile. Data that describes the characteristics of a user, group, resource, program, device, or remote location.

program temporary fix (PTF). For System i®, System p®, and System z® products, a fix that is tested by IBM and is made available to all customers. See also fix pack.

prompt. A component of an action that indicates that user input is required for a field before making a transition to an output screen.

property. A characteristic of an object that describes the object. A property can be changed or modified. Properties can describe an object name, type, value, or behavior, among other things.

protocol binding. A binding that enables the enterprise service bus to process messages independently of the communication protocol.

proxy. An application gateway from one network to another for a specific network application such as Telnet or FTP, for example, where a firewall proxy Telnet server performs authentication of the user and then lets the traffic flow through the proxy as if it were not there. Function is performed in the firewall and not in the client workstation, causing more load in the firewall.

proxy cluster. A group of proxy servers that distributes HTTP requests across the cluster.

proxy peer access point. A means of identifying the communication settings for a peer access point that cannot be accessed directly.

proxy server.

1. A server that acts as an intermediary for HTTP Web requests that are hosted by an application or a Web server. A proxy server acts as a surrogate for the content servers in the enterprise.

2. A server that receives requests intended for another server and that acts on behalf of the client (as the client's proxy) to obtain the requested service. A proxy server is often used when the client and the server are incompatible for direct connection. For example, the client is unable to meet the security authentication requirements of the server but should be permitted some services.

PTF. See program temporary fix.

public.

1. In object-oriented programming, pertaining to a class member that is accessible to all classes.

2. In the Java programming language, pertains to a method or variable that can be accessed by elements residing in other classes. (Sun)

QoS. See quality of service.

qualifier. A simple element that gives another generic compound or simple element a specific meaning. Qualifiers are used in mapping single or multiple occurrences. A qualifier can also be used to denote the namespace used to interpret the second part of the name, typically referred to as the ID.

quality of service (QoS). A set of communication characteristics that an application requires. Quality of Service (QoS) defines a specific transmission priority, level of route reliability, and security level.

query.

1. A request for information from a database based on specific conditions: for example, a request for a list of all customers in a customer table whose balances are greater than USD1000.

2. A reusable request for information about one or more model elements

read-through cache. A sparse cache that loads data entries by key as they are requested. When data cannot be found in the cache, the missing data is retrieved with the loader, which loads the data from the back-end data repository and inserts the data into the cache.

recursion. A programming technique in which a program or routine calls itself to perform successive steps in an operation, with each step using the output of the preceding step.

refresh pack. A cumulative collection of fixes that contains new functions. See also fix pack, interim fix.

region. A contiguous area of virtual storage that has common characteristics and that can be shared between processes.

replica. A server that contains a copy of the directory or directories of another server. Replicas back up servers in order to enhance performance or response times and to ensure data integrity.

replication. The process of maintaining a defined set of data in more than one location. Replication involves copying designated changes for one location (a source) to another (a target) and synchronizing the data in both locations.

resource.

1. A discrete asset, for example application suites, applications, business services, interfaces, endpoints, and business events.

2. A facility of a computing system or operating system required by a job, task, or running program. Resources include main storage, input/output devices, the processing unit, data sets, files, libraries, folders, application servers, and control or processing programs.

3. A person, piece of equipment, or material that is used to perform a task or a project. Each resource is a particular occurrence or example of a resource definition.

role.

1. A description of a function to be carried out by an individual or bulk resource, and the qualifications required to fulfill the function. In simulation and analysis, the term role is also used to refer to the qualified resources.

2. A job function that identifies the tasks that a user can perform and the resources to which a user has access. A user can be assigned one or more roles.

3. A logical group of principals that provides a set of permissions. Access to operations is controlled by granting access to a role.

4. In a relationship, a role determines the function and participation of entities. Roles capture structure and constraint requirements on participating entities and their manner of participation. For example, in an employment relationship, the roles are employer and employee.

root. The user name for the system user with the most authority.

run time. The time period during which a computer program is running.

runtime topology. A depiction of the momentary state of the environment.

scalability. The ability of a system to expand as resources, such as processors, memory, or storage, are added.

scope.

1. A specification of the boundary within which system resources can be used.

2. In Web services, a property that identifies the lifetime of the object serving the invocation request.

script. A series of commands, combined in a file, that carry out a particular function when the file is run. Scripts are interpreted as they are run.

scripting. A style of programming that reuses existing components as a base for building applications.

SDK. See software development kit.

Secure Sockets Layer (SSL). A security protocol that provides communication privacy. With SSL, client/server applications can communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery.

security administrator. The person who controls access to business data and program functions.

security token. A representation of a set of claims that are made by a client that can include a name, password, identity, key, certificate, group, privilege, and so on.

serialization. In object-oriented programming, the writing of data in sequential fashion to a communications medium from program memory.

serializer. A method for converting object data to another form such as binary or XML.

servant region. A contiguous area of virtual storage that is dynamically started as load increases and automatically stopped as load eases.

server. A software program or a computer that provides services to other software programs or other computers. See also host.

server cluster. A group of servers that are typically on different physical machines and have the same applications configured within them, but operate as a single logical server.

service level agreement (SLA). A contract between a customer and a service provider that specifies the expectations for the level of service with respect to availability, performance, and other measurable objectives.

servlet. A Java program that runs on a Web server and extends the server functions by generating dynamic content in response to Web client requests. Servlets are commonly used to connect databases to the Web.

session.

1. A logical or virtual connection between two stations, software programs, or devices on a network that allows the two elements to communicate and exchange data.

2. A series of requests to a servlet originating from the same user at the same browser.

3. In Java EE, an object used by a servlet to track user interaction with a Web application across multiple HTTP requests.

session affinity. A method of configuring applications in which a client is always connected to the same server. These configurations disable workload management after an initial connection by forcing a client request to always go to the same server.

setter method. A method whose purpose is to set the value of an instance or class variable. This capability allows another object to set the value of one of its variables.

shard. An instance of a partition. A shard can be a primary or replica.

shared lock. A lock that limits concurrently running application processes to read-only operations on database data.

shell script. A program, or script, that is interpreted by the shell of an operating system.

signer certificate. The trusted certificate entry that is typically in a truststore file.

silent installation. An installation that does not send messages to the console but instead stores messages and errors in log files. A silent installation can use response files for data input.

silent mode. A method for installing or uninstalling a product component from the command line with no GUI display. When using silent mode, you specify the data required by the installation or uninstallation program directly on the command line or in a file (called an option file or response file).

skeleton. Scaffolding for an implementation class.

SLA. See service level agreement.

software development kit (SDK). A set of tools, APIs, and documentation to assist with the development of software in a specific computer language or for a particular operating environment.

SQL. See Structured Query Language.

SQL query. A component of certain SQL statements that specifies a result table.

SSL. See Secure Sockets Layer.

SSL channel. A type of channel within a transport chain that associates a Secure Sockets Layer (SSL) configuration repertoire with the transport chain.

stack. An area in memory that typically stores information such as temporary register information, values of parameters, and return addresses of subroutines and is based on the principle of last in, first out (LIFO).

stand-alone. Independent of any other device, program, or system. In a network environment, a stand-alone machine accesses all required resources locally.

stand-alone server. A catalog service or container server that is managed from the operating system that starts and stops the server process.

static. A Java programming language keyword that is used to define a variable as a class variable.

string. In programming languages, the form of data used for storing and manipulating text.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

subclass. In Java, a class that is derived from a particular class, through inheritance.

subquery. In SQL, a subselect used within a predicate, for example, a select-statement within the WHERE or HAVING clause of another SQL statement.

synchronize. To add, subtract, or change one feature or artifact to match another.

synchronous process. A process that starts by invoking a request-response operation. The result of the process is returned by the same operation.

synchronous replica. A shard that receives updates as part of the transaction on the primary shard to guarantee data consistency, which can increase the response time compared with an asynchronous replica.

syntax. The rules for the construction of a command or statement.

systems analyst. A specialist who is responsible for translating business requirements into system definitions and solutions.

TCP. See Transmission Control Protocol.

TCP channel. A type of channel within a transport chain that provides client applications with persistent connections within a local area network (LAN).

TCP/IP. See Transmission Control Protocol/Internet Protocol.

TCP/IP monitoring server. A runtime environment that monitors all requests and responses between a Web browser and an application server, as well as TCP/IP activity.

thin application client. A lightweight, downloadable Java application run time capable of interacting with enterprise beans.

thin client. A client that has little or no installed software but has access to software that is managed and delivered by network servers that are attached to it. A thin client is an alternative to a full-function client such as a workstation.

thread. A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

thread contention. A condition in which a thread is waiting for a lock or object that another thread holds.

threshold. A setting that applies to an interrupt in a simulation that defines when a process simulation should be halted based on a condition existing for a specified proportion of occurrences of some event.

throughput. The measure of the amount of work performed by a device, such as a computer or printer, over a period of time, for example, number of jobs per day.

time to live. The time interval in seconds that an entry can exist in the cache before that entry is discarded.

timeout. A time interval that is allotted for an event to occur or complete before operation is interrupted.

timer. A task that produces output at certain points in time.

timing constraint. A specialized validation action used to measure the duration of a method call or a sequence of method calls.

Tivoli Performance Viewer. A Java client that retrieves the Performance Monitoring Infrastructure (PMI) data from an application server and displays it in various formats.

token.

1. A marker used to track the current state of a process instance during a simulation run.

2. A particular message or bit pattern that signifies permission or temporary control to transmit over a network.

topology. The physical or logical mapping of the location of networking components or nodes within a network. Common network topologies include bus, ring, star, and tree.

transaction. A process in which all of the data modifications that are made during a transaction are either committed together as a unit or rolled back as a unit.

Transmission Control Protocol (TCP). A communication protocol used in the Internet and in any network that follows the Internet Engineering Task Force (IETF) standards for internetwork protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in interconnected systems of such networks.

Transmission Control Protocol/Internet Protocol (TCP/IP). An industry-standard, nonproprietary set of communication protocols that provides reliable end-to-end connections between applications over interconnected networks of different types.

truststore file. A key database file that contains the public keys for a trusted entity.

type.

1. In Java programming, a class or interface.

2. In a WSDL document, an element that contains data type definitions using some type system (such as XSD).

UDDI. See Universal Description, Discovery, and Integration.

Uniform Resource Identifier (URI).

1. A compact string of characters for identifying an abstract or physical resource.

2. A unique address that is used to identify content on the Web, such as a page of text, a video or sound clip, a still or animated image, or a program. The most common form of URI is the Web page address, which is a particular form or subset of URI called a Uniform Resource Locator (URL). A URI typically describes how to access the resource, the computer that contains the resource, and the name of the resource (a file name) on the computer.

Uniform Resource Locator (URL). The unique address of an information resource that is accessible in a network such as the Internet. The URL includes the abbreviated name of the protocol used to access the information resource and the information used by the protocol to locate the information resource.

Uniform Resource Name (URN). A name that uniquely identifies a Web service to a client.

Universal Description, Discovery, and Integration (UDDI). A set of standards-based specifications that enables companies and applications to quickly and easily find and use Web services over the Internet.

Universally Unique Identifier (UUID). The 128-bit numerical identifier that is used to ensure that two components do not have the same identifier.

UNIX System Services. An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specifications and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

upgradeable lock. A lock that identifies the intent to update a cache entry when using a pessimistic lock.

upstream. Pertaining to the direction of the flow, which is from the start of the process (upstream) toward the end of the process (downstream).

URI. See Uniform Resource Identifier.

URL. See Uniform Resource Locator.

URL scheme. A format that contains another object reference.

URN. See Uniform Resource Name.

UUID. See Universally Unique Identifier.

variable. A representation of a changeable value.

version. A separately licensed program that typically has significant new code or new function.

virtual host. A configuration enabling a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine.

virtual machine. An abstract specification for a computing device that can be implemented in different ways in software and hardware.

virtualization. A technique that encapsulates the characteristics of resources from the way in which other systems interact with those resources.

waiter. A thread waiting for a connection.

WAR. See Web archive.

WCCM. See WebSphere Common Configuration Model.

Web archive (WAR). A compressed file format, defined by the Java EE standard, for storing all the resources required to install and run a Web application in a single file. See also enterprise archive.

Web browser. A client program that initiates requests to a Web server and displays the information that the server returns.

Web component. A servlet, JavaServer Pages (JSP) file, or a HyperText Markup Language (HTML) file. One or more Web components make up a Web module.

Web container. A container that implements the Web component contract of the Java EE architecture. (Sun)

Web container channel. A type of channel within a transport chain that creates a bridge in the transport chain between an HTTP inbound channel and a servlet or JavaServer Pages (JSP) engine.

Web crawler. A type of crawler that explores the Web by retrieving a Web document and following the links within that document.

Web server. A software program that is capable of servicing Hypertext Transfer Protocol (HTTP) requests.

Web server plug-in. A software module that supports the Web server in communicating requests for dynamic content, such as servlets, to the application server.

Web server separation. A topology where the Web server is physically separated from the application server.

Web site. A related collection of files available on the Web that is managed by a single entity (an organization or an individual) and contains information in hypertext for its users. A Web site often includes hypertext links to other Web sites.

WebSphere. An IBM brand name that encompasses tools for developing e-business applications and middleware for running Web applications.

WebSphere Common Configuration Model (WCCM). A model that provides for programmatic access to configuration data.

what you see is what you get (WYSIWYG). A capability of an editor to continually display pages exactly as they will be printed or otherwise rendered.

while loop. A loop that repeats the same sequence of activities as long as some condition is satisfied. The while loop tests its condition at the beginning of every loop. If the condition is false from the start, the sequence of activities contained in the loop never runs.

WLM. See Workload Manager.

workload management. The optimization of the distribution of incoming work requests to the application servers, enterprise beans, servlets and other objects that can effectively process the request.

Workload Manager (WLM). A component of z/OS that provides the ability to run multiple workloads at the same time within one z/OS image or across multiple images.

workspace.

1. A directory on disk that contains all project files, as well as information such as preferences.

2. A temporary repository of configuration information that administrative clients use.

3. In Eclipse, the collection of projects and other resources that the user is currently developing in the workbench. Metadata about these resources resides in a directory on the file system; the resources might reside in the same directory.

write-behind cache. A cache that asynchronously writes each write operation to the database using a loader.

write-through cache. A cache that synchronously writes each write operation to the database using a loader.

WYSIWYG. See what you see is what you get.

X/Open XA. The X/Open Distributed Transaction Processing XA interface. A proposed standard for distributed transaction communication. The standard specifies a bidirectional interface between resource managers that provide access to shared resources within transactions, and between a transaction service that monitors and resolves transactions.

XA. A bidirectional interface between one or more resource managers that provide access to shared resources and a transaction manager that monitors and resolves transactions.

XML. See Extensible Markup Language.

z/OS. An IBM mainframe operating system that uses 64-bit real storage.

zone-based support. A function that enables rules-based shard placement to improve grid availability by placing shards across different data centers, whether on different floors or even in different buildings or geographies.

Notices

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

Index

Printed in USA