{brandname} 10.0 Glossary

The {brandname} community

Table of Contents

This glossary aims to clarify some of the terms frequently encountered in {brandname}'s User Guide, Getting Started Guide, FAQs, etc.

Chapter 1. 2-phase commit

2-phase commit protocol (2PC) is a consensus protocol used for atomically commit or rollback distributed transactions.

More resources

- Wikipedia article

Chapter 2. Atomicity, Consistency, Isolation, Durability (ACID)

According to Wikipedia, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

More resources

- Wikipedia

Chapter 3. Basically Available, Soft-state, Eventually-consistent (BASE)

BASE, also known as Eventual Consistency, is seen as the polar opposite of *ACID*, properties seen as desirable in traditional database systems.

BASE essentially embraces the fact that true consistency cannot be achieved in the real world, and as such cannot be modelled in highly scalable distributed systems. BASE has roots in Eric Brewer's *CAP Theorem*, and eventual consistency is the underpinning of any distributed system that aims to provide high availability and partition tolerance.

{brandname} has traditionally followed ACID principles as far as possible, however an eventually consistent mode embracing BASE is on the roadmap.

More resources

- A good article on ACM compares BASE versus ACID.
- An excellent talk on eventual consistency and BASE in Riak is also available on InfoQ.

Chapter 4. Consistency, Availability and Partition-tolerance (CAP) Theorem

Made famous by Eric Brewer at UC Berkeley, this is a theorem of distributed computing that can be simplified to state that one can only practically build a distributed system exhibiting any two of the three desirable characteristics of distributed systems, which are: Consistency, Availability and Partition-tolerance (abbreviated to CAP). The theorem effectively stresses on the unreliability of networks and the effect this unreliability has on predictable behavior and high availability of dependent systems.

{brandname} has traditionally been biased towards Consistency and Availability, sacrificing Partition-tolerance. However, {brandname} does have a Partition-tolerant, eventually-consistent mode in the pipeline. This optional mode of operation will allow users to tune the degree of consistency they expect from their data, sacrificing partition-tolerance for this added consistency.

More resources

- The theorem is well-discussed online, with many good resources to follow up on, including this document.
- A more recent article by Eric Brewer himself appears on InfoQ a modern analysis of the theorem .

Chapter 5. Consistent Hash

A technique of mapping keys to servers such that, given a stable cluster topology, any server in the cluster can locate where a given key is mapped to with minimal computational complexity.

Consistent hashing is a purely algorithmic technique, and doesn’t rely on any metadata or any network broadcasts to "search" for a key in a cluster. This makes it extremely efficient to use.

More resources

- Wikipedia

Chapter 6. Data grid

A data grid is a cluster of (typically commodity) servers, normally residing on a single local-area network, connected to each other using IP based networking. Data grids behave as a single resource, exposing the aggregate storage capacity of all servers in the cluster. Data stored in the grid is usually partitioned, using a variety of techniques, to balance load across all servers in the cluster as evenly as possible. Data is often redundantly stored in the grid to provide resilience to individual servers in the grid failing i.e. more than one copy is stored in the grid, transparently to the application.

Data grids typically behave in a peer-to-peer fashion. {brandname}, for example, makes use of JGroups as a group communication library and is hence biased towards a peer-to-peer design. Such design allows {brandname} to exhibit self-healing characteristics, providing service even when individual servers fail and new nodes are dynamically added to the grid.

{brandname} also makes use of TCP and optionally UDP network protocols, and can be configured to make use of IP multicast for efficiency if supported by the network.

Chapter 7. Deadlock

A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

Chapter 8. Distributed Hash Table (DHT)

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Chapter 9. Externalizer

An *Externalizer* is a class that knows how to marshall a given object type to a byte array, and how to unmarshall the contents of a byte array into an instance of the object type. Externalizers are effectively an {brandname} extension that allows users to specify how their types are serialized. The underlying {brandname} marshalling infrastructure builds on JBoss Marshalling , and offers efficient payloads and stream caching. This provides much better performance than standard Java serialization.

More resources

- Plug your own Externalizer implementation into {brandname}

Chapter 10. Hot Rod

Hot Rod is the name of {brandname}'s custom TCP client/server protocol which was created in order to overcome the deficiencies of other client/server protocols such as Memcached. HotRod, as opposed to other protocols, has the ability of handling failover on an {brandname} server cluster that undergoes a topology change. To achieve this, the Hot Rod regularly informs the clients of the cluster topology.

Hot Rod enables clients to do smart routing of requests in partitioned, or distributed, {brandname} server clusters. This means that Hot Rod clients can determine the partition in which a key is located and communicate directly with the server that contains the key. This is made possible by {brandname} servers sending the cluster topology to clients, and the clients using the same consistent hash as the servers.

More resources

- Information about the protocol
- Starting a Hot Rod server
- Hot Rod client libraries

Chapter 11. In-memory data grid

An in-memory data grid (IMDG) is a special type of data grid. In an IMDG, each server uses its main system memory (RAM) as primary storage for data (as opposed to disk-based storage). This allows for much greater concurrency, as lock-free STM techniques such as compare-and-swap can be used to allow hardware threads accessing concurrent datasets. As such, IMDGs are often considered far better optimized for a multi-core and multi-CPU world when compared to disk-based solutions. In addition to greater concurrency, IMDGs offer far lower latency access to data (even when compared to disk-based data grids using solid state drives).

The tradeoff is capacity. Disk-based grids, due to the far greater capacity of hard disks, exhibit two (or even three) orders of magnitude greater capacity for the same hardware cost.

Chapter 12. Isolation level

Isolation is a property that defines how/when the changes made by one operation become visible to other concurrent operations. Isolation is one of the *ACID* properties.

{brandname} ships with REPEATABLE_READ and READ_COMMITTED isolation levels, the latter being the default.

Chapter 13. JTA synchronization

A Synchronization is a listener which receives events relating to the transaction lifecycle. A Synchronization implementor receives two events, *before completion* and *after completion* . Synchronizations are useful when certain activities are required in the case of a transaction completion; a common usage for a Synchronization is to flush an application's caches.

Chapter 14. Livelock

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Chapter 15. Memcached

Memcached is an in-memory caching system, often used to speed-up database-driven websites. Memcached also defines a text based, client/server, caching protocol, known as the Memcached protocol {brandname} offers a server which speaks the Memcached protocol, allowing Memcached itself to be replaced by {brandname}. Thanks to {brandname}'s clustering capabilities, it can offer data failover capabilities not present in original Memcached systems.

More resources

- {brandname}'s Memcached Server
- The memcached website

Chapter 16. Multiversion Concurrency Control (MVCC)

Multiversion concurrency control is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

More resources

- Wikipedia

Chapter 17. Near Cache

A technique for caching data in the client when communicating with a remote cache, for example, over the *Hot Rod* protocol. This technique helps minimize remote calls to retrieve data.

Chapter 18. Network partition

Network partitions happens when multiple parts of a cluster become separated due to some type of network failure, whether permanent or temporary. Often temporary failures heal spontaneously, within a few seconds or at most minutes, but the damage that can occur during a network partition can lead to inconsistent data. Closely tied to Brewer's CAP theorem, distributed systems choose to deal with a network partition by either sacrificing availability (either by shutting down or going into read-only mode) or consistency by allowing concurrent and divergent updates to the same data.

Network partitions are also commonly known as a *Split Brain*, after the biological condition of the same name.

For more detailed discussion, see this blog post.

Chapter 19. NoSQL

A NoSQL database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability. NoSQL databases are often highly optimized key–value stores intended for simple retrieval and appending operations, with the goal being significant performance benefits in terms of latency and throughput. NoSQL databases are finding significant and growing industry use in big data and real-time web applications.

Chapter 20. Optimistic locking

Optimistic locking is a concurrency control method that assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

Chapter 21. Pessimistic locking

A lock is used when multiple threads need to access data concurrently. This prevents data from being corrupted or invalidated when multiple threads try to modify the same item of data. Any single thread can only modify data to which it has applied a lock that gives them exclusive access to the record until the lock is released. However, pessimistic locking isn't ideal from a throughput perspective, as locking is expensive and serializing writes may not be desired. *Optimistic locking* is often seen as a preferred alternative in many cases.

Chapter 22. READ COMMITTED

READ_COMMITTED is one of two isolation levels the {brandname}'s locking infrastructure provides (the other is REPEATABLE_READ). Isolation levels have their origins in relational databases.

In {brandname}, READ_COMMITTED works slightly differently to databases. READ_COMMITTED says that "data can be read as long as there is no write", however in {brandname}, reads can happen anytime thanks to *MVCC*. MVCC allows writes to happen on copies of data, rather than on the data itself. Thus, even in the presence of a write, reads can still occur, and all read operations in {brandname} are non-blocking (resulting in increased performance for the end user). On the other hand, write operations are exclusive in {brandname}, (and so work the same way as READ_COMMITTED does in a database).

With READ_COMMITTED, multiple reads of the same key within a transaction can return different results, and this phenomenon is known as non-repeatable reads. This issue is avoided with REPETEABLE_READ isolation level.

Chapter 23. Relational Database Management System (RDBMS)

A relational database management system (RDBMS) is a database management system that is based on the relational model. Many popular databases currently in use are based on the relational database model.

Chapter 24. REPEATABLE READ

REPEATABLE_READ is one of two isolation levels the {brandname}'s locking infrastructure provides (the other is READ_COMMITTED). Isolation levels have their origins in relational databases.

In {brandname}, REPEATABLE_READ works slightly differently to databases. REPEATABLE_READ says that "data can be read as long as there are no writes, and vice versa". This avoids the non-repeatable reads phenomenon, because once data has been written, no other transaction can read it, so there's no chance of re-reading the data and finding different data.

Some definitions of REPEATABLE_READ say that this isolation level places shared locks on read data; such lock could not be acquired when the entry is being written. However, {brandname} has an MVCC concurrency model that allows it to have non-blocking reads. {brandname} provides REPEATABLE_READ semantics by keeping the previous value whenever an entry is modified. This allows {brandname} to retrieve the previous value if a second read happens within the same transaction, but it allows following phenomena:

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                                    tx2.begin()
Thread2:                                    cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                                    cache.get("B") // returns 2
Thread2:                                    tx2.commit()
```

By default, {brandname} uses REPEATABLE_READ as isolation level.

Chapter 25. Representational State Transfer (ReST)

ReST is a software architectural style that promotes accessing resources via a uniform generic interface. HTTP is an implementation of this architecture, and generally when ReST is mentioned, it refers to ReST over HTTP protocol. When HTTP is used, the uniform generic interface for accessing resources is formed of GET, PUT, POST, DELETE and HEAD operations.

{brandname}'s ReST server offers a ReSTful API based on these HTTP methods, and allow data to be stored, retrieved and deleted.

More resources

- The {brandname} REST Server

Chapter 26. Split brain

A colloquial term for a *network partition*. See *network partition* for more details.

Chapter 27. Structured Query Language (SQL)

SQL is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS). Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control.

Chapter 28. Write-behind

Write-behind is a cache store update mode. When this mode is used, updates to the cache are asynchronously written to the cache store. Normally this means that updates to the cache store are not performed in the client thread.

An alternative cache store update mode is *write-through*.

More resources

- {brandname} User guide

Chapter 29. Write skew

In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other. Were the system serializable, such an anomaly would be impossible, as either T1 or T2 would have to occur "first", and be visible to the other. In contrast, snapshot isolation such as REPEATABLE_READ and READ_COMMITTED permits write skew anomalies.

{brandname} can detect write skews and can be configured to roll back transactions when write skews are detected.

Chapter 30. Write-through

Write-through is a cache store update mode. When this mode is used, clients update a cache entry, e.g. via a Cache.put() invocation, the call will not return until {brandname} has updated the underlying cache store. Normally this means that updates to the cache store are done in the client thread.

An alternative mode in which cache stores can be updated is *write-behind*.

More resources

- {brandname} User guide

Chapter 31. XA resource

An XA resource is a participant in an XA transaction (also known as a distributed transaction). For example, given a distributed transaction that operates over a database and {brandname}, XA defines both {brandname} and the database as XA resources.

Java's API for XA transactions is JTA and XAResource is the Java interface that describes an XA resource.