Clustering and Partitioning

A cluster in Ignite is a set of interconnected nodes where server nodes (Node 0, Node 1, etc. in Figure 1) are organized in a logical ring and client nodes (applications) are connected to them. Both server and client nodes can execute all the APIs supported in Ignite. The main difference between the two is that client nodes do not store data.

Data that are stored on the server nodes are represented as key-value pairs. The pairs in their turn are located in specific partitions which belong to individual Ignite caches as shown in Figure 2.
To ensure data consistency and comply with the high-availability principle, server nodes are capable of storing a primary as well as backup copies of data. Basically, there is always a primary copy of a partition with all its key-value pairs in the cluster and there may be 0 or more backup copies of the same partition depending upon the configuration parameters.

Every cluster node (server or client) is aware of all primary and backup copies of every partition. This information is collected and broadcast to all the nodes from a coordinator (the oldest server node) via internal partition map exchange messages.

However, all the data related requests/operations (get, put, SQL, etc.) go to primary partitions except for some read operations when CacheConfiguration.readFromBackup is enabled. If it's an update operation (put, INSERT, UPDATE) then Ignite ensures that both the primary and backup copies are updated and stay in a consistent state.

Transactions

This section dives into the details of the Ignite transactional protocol. High-level principles and features are described in the Ignite technical documentation.

Two-Phase Commit Protocol

A single transaction in distributed systems usually spans across several server nodes which imposes additional requirements for data consistency. For instance, it is obligatorily to detect and handle situations when a transaction was not fully committed due to a partial outage or cluster nodes loss. Ignite relies on two-phase commit for handling this and many other situations in order to ensure data consistency cluster-wide.

As the protocol name suggests, a transaction is executed in two phases. The "prepare" phase goes first, as shown in Figure 3.
Figure 3.

1. Transaction coordinator (also known as near node or application that runs a transaction) sends a "prepare" message (Step 1) to all primary nodes participating in the transaction.

2. The primary nodes forward the message to nodes that hold a backup copy of data if any (Step 2) and acquire all the required locks.

3. The primary nodes acknowledge (Step 4) that all the locks are acquired and they are ready to commit the transaction.

Next, the transaction coordinator executes the second phase by sending a "commit" message, as shown in Figure 4.
Once the backup and primary copies are updated, the transaction coordinator receives an acknowledgement and assumes that the transaction is finished.

This is how the 2-phase commit works in a nutshell. Below we will see how the protocol tolerates failures, distinguishes pessimistic and optimistic transactions and does many other things.

**Near Node and Remote Node**

The transaction coordinator is also known as a near node in the Ignite community and committers. The transaction coordinator initiates a transaction, tracks its state, sends over "prepare" and "commit" messages, and orchestrates the overall transaction process. Usually, the coordinator is a client node that connects applications to the cluster. An application triggers `tx.call()`, `cache.put()`, `cache.get()`, and `tx.commit()` methods and the client node takes care of the rest as shown in Figure 5.
In addition to the transaction coordinator, the transaction protocol defines remote nodes which are server nodes that keep a part of the data being accessed or updated inside of the transaction. Internally, every server node maintains a Distributed Hash Table (DHT) for partitions it owns. The DHT helps to look up partition owners (primary and backups) efficiently from any cluster node including the transaction coordinator. Note that the data are stored in pages that are arranged by B+Tree (refer to memory architecture documentation for more details).

**Locking Modes and Isolation Levels**

This section elaborates on locking modes and isolation levels supported in Ignite. A high-level overview is also available in the Ignite technical documentation.

**Pessimistic Locking**

In multi-user applications, different users can modify the same data simultaneously. To deal with reads and updates of the same data sets happening in parallel, transaction subsystems of products such as Ignite implement optimistic and pessimistic locking. In the pessimistic mode, an application will acquire locks for all the data it plans to change and will apply the changes after all the locks are owned exclusively while in the optimistic mode lock acquisition is postponed to a later phase when a transaction is being committed.

Lock acquisition time also depends on the type of isolation level. Let's start with the review of isolation levels in conjunction with the pessimistic mode.

In pessimistic and read committed mode, locks are acquired before the changes brought by write operations such (as put or putAll) are applied, as shown in Figure 6.
However, Ignite never obtains locks for read operations (such as get or getAll) in this mode which might be inappropriate for some of the use cases. To ensure the locks are acquired prior to every read or write operation, use either repeatable read or serializable isolation level in Ignite, as shown in Figure 7.

The pessimistic mode holds locks until the transaction is finished and prevents access to locked data from other transactions. Optimistic transactions in Ignite might increase the throughput of an application by lowering contention among transactions by moving lock acquisition to a later phase.

Optimistic Locking

In optimistic transactions, locks are acquired on primary nodes during the "prepare" phase, then promoted to backup nodes and released once the transaction is committed. Depending on an isolation level, if Ignite detects that a version of an entry has been changed since the time it was requested by a transaction, then the transaction will fail at the "prepare" phase and it will be up to an application to decide whether to restart the transaction or not. This is exactly how optimistic and serializable transactions (also known as deadlock-free transactions) work in Ignite, as shown in Figure 8.
On the other hand, repeatable read and read committed optimistic transactions never check if a version of an entry is changed. This mode (Figure 9) might bring extra performance benefits but does not give any atomicity guarantees and, thus, is rarely used in practice.
Now let's review the entire lifecycle of a transaction in Ignite. We will assume that the cluster is stable and no outages happen.

In Figure 10, the transaction starts once `tx.start()` (Step 1) method is called by the application which results in the creation (Step 2) of a structure for transaction context management (also known as `IgniteInternalTx`). Additionally, the following happens on the near node side:

- A unique transaction identifier is generated.
- The start time of the transaction is recorded.
- Current topology version/state is recorded.

Next, the transaction status is set to "active" and Ignite starts executing read/write operations that are a part of the transaction using the rules of either optimistic or pessimistic mode and specific isolation levels.

**Transaction Commit**

When the application executes the `tx.commit()` method (Step 9 in Figure 10) the near node (transaction coordinator) initiates the 2-phase commit protocol by preparing the "prepare" message enclosing information about the transaction's context into it.

As a part of the "prepare" phase, every primary node receives information about all updated or new key-value pairs and about the order the locks have to be acquired (the latter depends on a combination of locking modes and isolation levels).

The primary nodes in their turn perform the following in response to the "prepare" message:

- Check that a version of the cluster topology recorded in the transaction's context matches the current topology version.
- Obtain all the required locks.
- Create a DHT context for the transaction and store all the necessary data therein.
- Depending upon the cache configuration parameters, wait or skip waiting while backup nodes confirm that the "prepare" phase is over.
- Inform the near node that it's time to execute the "commit" phase.

After that, the near node sends the "commit" message, waits for an acknowledgment and moves the transaction status to "committed".

**Transaction Rollback**

If the transaction was rolled back (`tx.rollback()` is called by the application), then, in the pessimistic mode, Ignite would be required to release all the acquired locks and delete the transaction's context. In the optimistic mode the locks are acquired when `tx.commit()` is called by the application, therefore, Ignite would simply clean out the transaction's context on the transaction coordinator (near node).
Transaction Timeout

Ignite allows setting a timeout for a transaction. If transaction’s execution time exceeds the timeout, then the transaction will be aborted.

In the pessimistic mode, the timeout is compared to the current total execution time every time an entry lock is acquired and when the “prepare” phase is triggered. In the optimistic mode, the timeout is compared only in the “prepare” phase. Figure 11 shows a summary.

Timeout Lock

If the total execution time has exceeded the timeout on any of the participating nodes, then a primary node, where the timeout elapsed, sets a special flag instructing the transaction coordinator to initiate a transaction cancellation.

Failover and Recovery

The section explains how Ignite tackles failover situations or outages that might happen while transactions are being executed.

Backup Node Failure
The simplest failure scenario is when a backup node fails on either the "prepare" or the "commit" phase. Nothing has to be done by the Ignite transactional subsystem. Transaction modifications will be applied to the remaining primary and backup nodes (Figure 12) and a new backup node for missing partitions will be elected after the transaction is over and that will not preload all the up-to-date data from a respective primary node.

Primary Node Failures on Prepare Phase
If a primary node failed before or on the "Prepare" phase, then the transaction coordinator raises an exception (Figure 13) and it's up to the application to decide what to do next - restart the transaction or process this exception differently.

Primary Node Failures on Commit Phase

If a primary node failed after the "prepare" phase, then the transaction coordinator will be waiting for an extra NodeFailureDetection response from respective backup nodes (Figure 14).

![Primary Node Failures on Commit](image)

Figure 14.

Once the backup nodes detect the failure they will send a message to the transaction coordinator confirming that they successfully committed the changes and no data loss happened because there is still an extra backup copy available for application usage.

Next, the transaction coordinator finishes the transaction, the topology is changed (due to the primary node loss) and the cluster will elect a new primary for the partitions that were stored on the previous one.

Transaction Coordinator Failure

Handling of transaction coordinator failures is a bit trickier because every remote node (primary and backup) is aware of the transaction's context related to it and doesn't know the overall transaction state. It is also possible that some of the nodes have already received "commit" message while the others haven't, as shown in Figure 15.
To solve this situation, the primary nodes exchange internal status with each other to find out the overall transaction state. For instance, if one of the nodes responds that it hasn’t received a “commit” message, then the transaction will be rolled back globally.

Transaction guarantees schematically

<table>
<thead>
<tr>
<th>X - Concurrency</th>
<th>Y - Isolation</th>
<th>Pessimistic</th>
<th>Pessimistic</th>
<th>Optimistic</th>
<th>Optimistic</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Read_commit</td>
<td>Repeatable_read</td>
<td>Read_commit</td>
<td>Repeatable_read</td>
<td>Serializable</td>
</tr>
</tbody>
</table>

// Open tx with X concurrency
// and Y isolation
tx = txStart(X, Y) {

// Read data inside tx.
Account acct1 = cache.get(acctId1);
Account acct2 = cache.get(acctId2);

// Make changes to the data.
acct1.update(100);
acct2.update(200);

// Store updated data in cache.
acct.put(acctId1, acct1);
cache.put(acctId2, acct2);

// Commit the tx.
tx.commit();
}

* Even though inconsistent read is possible within the transaction logic, it will be validated during commit phase and the whole transaction will be rolled back.

** Even though the locks are acquired on commit phase, in OPTIMISTIC SERIALIZABLE mode the locks are not acquired sequentially. Only individual locks are acquired in parallel. This means that multiple entries are not locked together and deadlocks are not possible.