Apache Ignite Transactions Architecture

Version

This paper focuses on the current implementation of the transaction mechanism Apache Ignite.

We:

1. Very briefly recall the main points of architecture Ignite
2. Talk about protocol 2-phase commit transactions
3. to find out what lock modes supported Ignite
4. Consider how the transaction is
5. processed How a situation where the "lost" nodes at different stages of the transaction

To begin, consider the definition and basic properties of Ignite.

1. Introduction to Apache Ignite

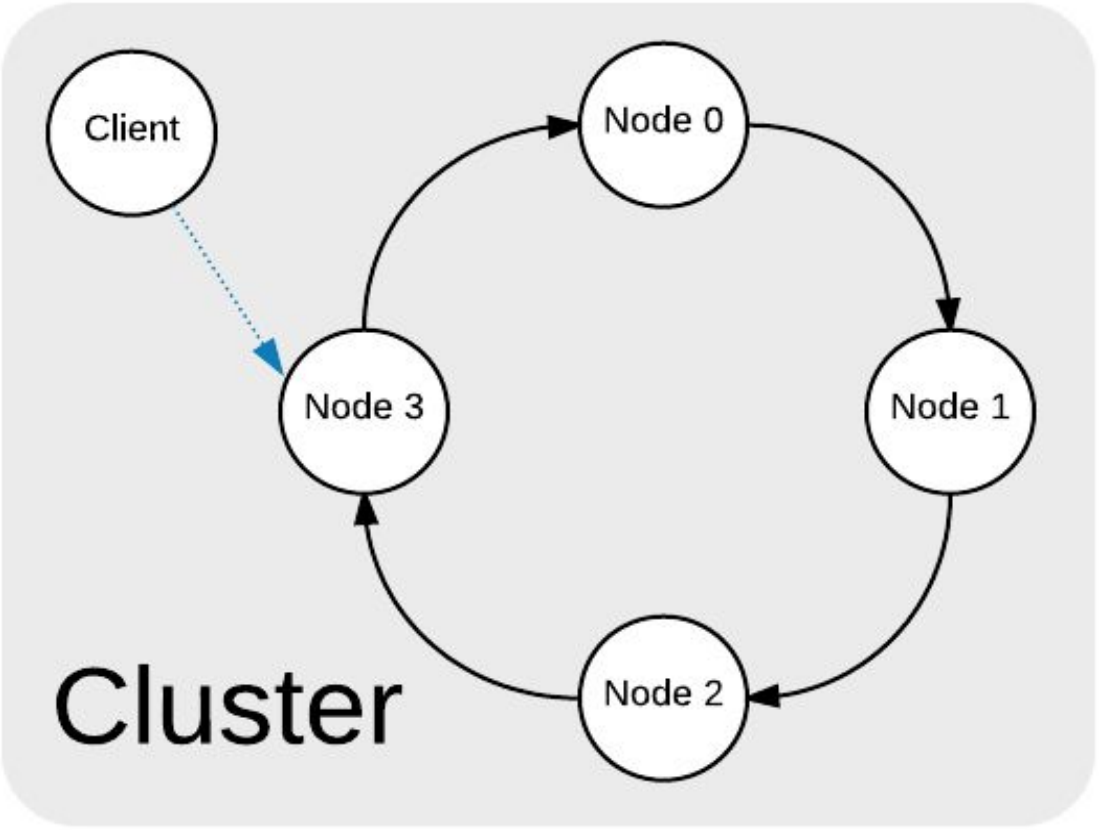
1.1. Apache Ignite

Ignite for topical at the moment is the following definition: "Ignite - a distributed, memory-centric database."
As a distributed system, it has received key-value pairs to store data. All algorithms it developed and optimized for parallel computing. How the DBMS allows it to experience to experience a reboot, and provides support for distributed indexes and SQL queries on the full set of data in memory and on disk.

More fully Ignite, a system to:

1. create an elastic, horizontally scalable Data Grid scaling thousands of nodes, with data distribution through affinity function for reducing excess movement data when adding or removing nodes.
2. use Data Grid created to perform parallel calculations in Compute Grid. And the code is executed directly on the node, which contains the required data.
3. Durable Memory used to store data not only in memory, but also on the disc.
4. run distributed SQL ANSI-99 queries across the data set, which are both in memory and on disk any of the nodes.
5. build, train, test, Machine Learning algorithms and models, specially optimized to work on together distributed data.

1.2. Cluster / Cache / Partitions



Ignite A cluster is a set ring integrated into the server and connected to it, client nodes.
And server and client nodes may perform completely all the tasks Ignite API, start a transaction, to participate in the calculations and loading.
The main difference between the two is that the client node does not store data, and they require a server node to connect to the cluster.

For the user, the main value of the cluster is the ability to store data and perform computations on them.
Data is stored as pairs of key - value, and the method combining such pairs with common settings protrude caches, which are combined into a group of caches.

When stored in the cache large amounts of data, it is divided into several parts partitions - partition.

To ensure data security, in addition to the main (Primary) copies created its backup (Backup) copies.
Each node in the cluster (both server and client) "knows" about the All Primary and Backup copies of the partition. This information is collected and sent to the coordinator of the cluster in Partition Map Exchange message to all nodes.
For the user, all work with cache (and put, and get operations) is performed only through the Primary partition (except where it has been defined with the execution of Backup partition read operation).
If Primary section is not available (e.g. node is overloaded), the transaction must wait until the topology change (because one of the nodes has gone out of service) and the definition of a new node with Primary partition. After this transaction will try to perform the operation once again.

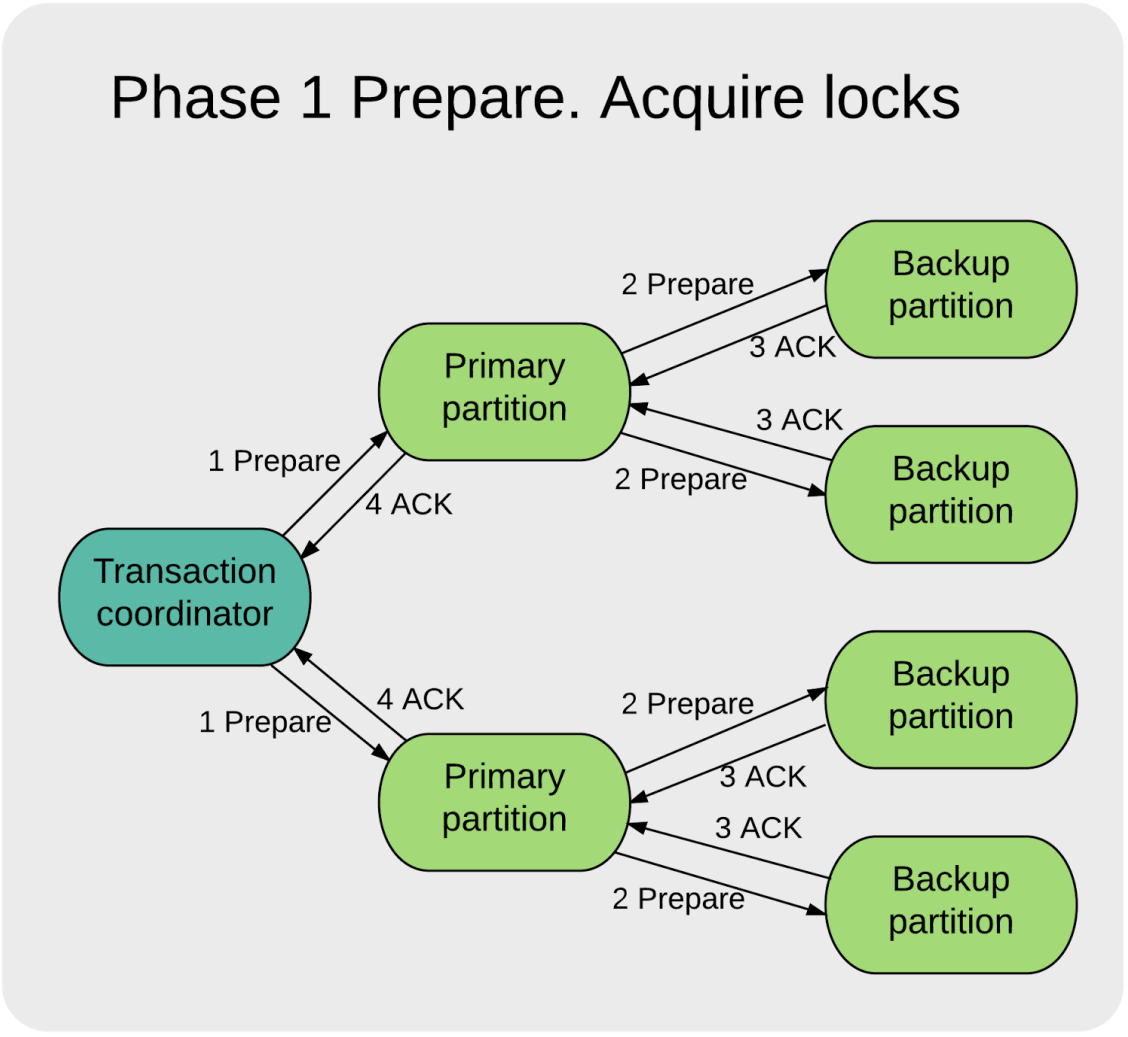
2. Transactions

2.1. Two-Phase-Commit Protocol

In distributed systems, a single transaction may involve data located on different nodes, which imposes a (for compliance with the principles of atomic and consistency) additional requirements. It is necessary to monitor and handle the situation correctly when changes are applied only part of nodes. Or when a portion of units out of service.

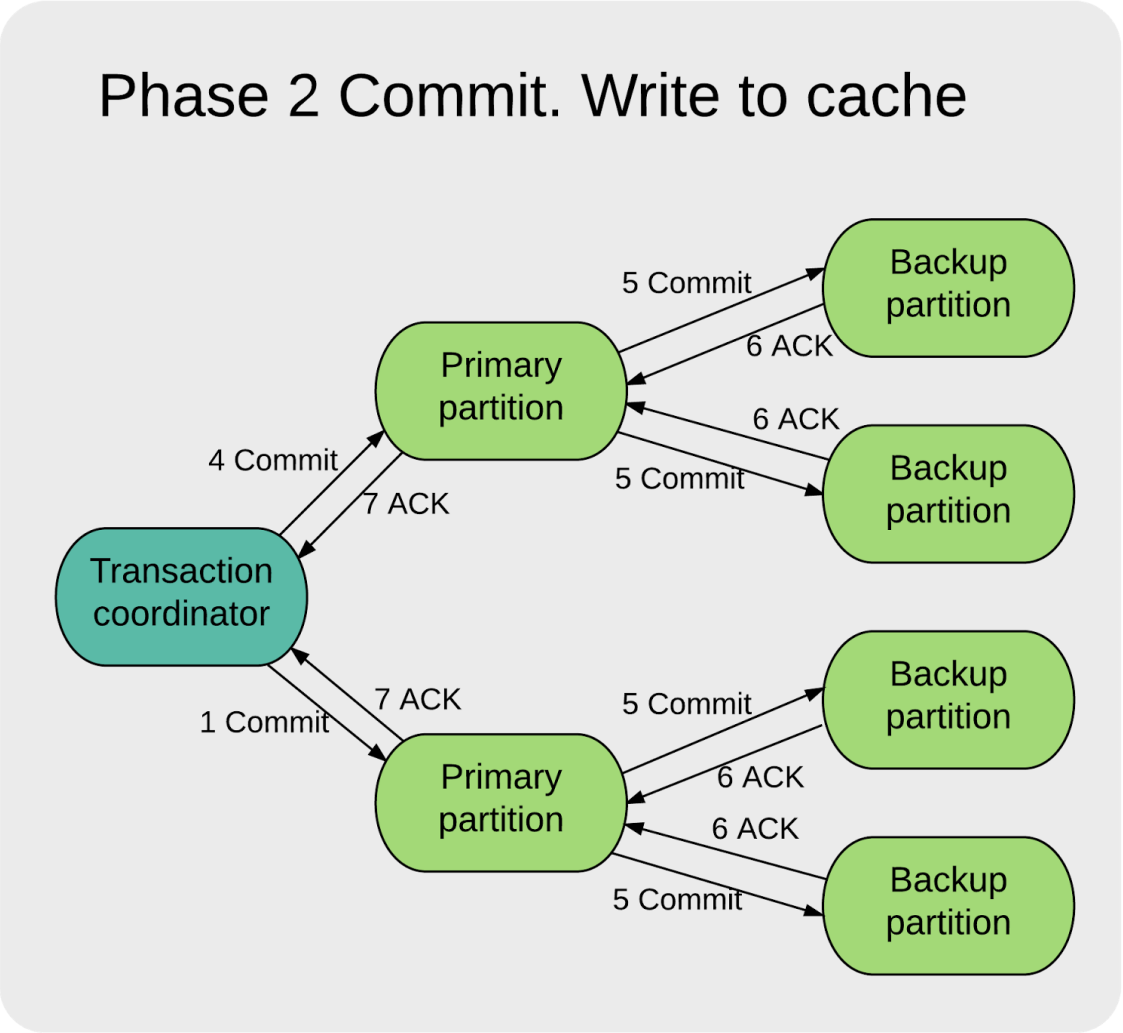
The traditional algorithm for solving such problems algorithm performs 2-phase commit changes. In Orakle such transactions are called distributed (distributed), they run in multiple bases connected through dblink.

Protocol 2-phase commit, as its name implies, is performed in two steps.



1. Coordinator transaction, NearNode terminology Ignite Primary sends to all nodes participating in the transaction, "Prepare" message.
After receiving such a message, the node synchronization is trying to "grab" all necessary locks and, if successful, sends the message confirming their participation in the transaction.

2. Coordinator, received confirmation from all the participants, sends "Commit".
After that, the node "fixes" the transaction.

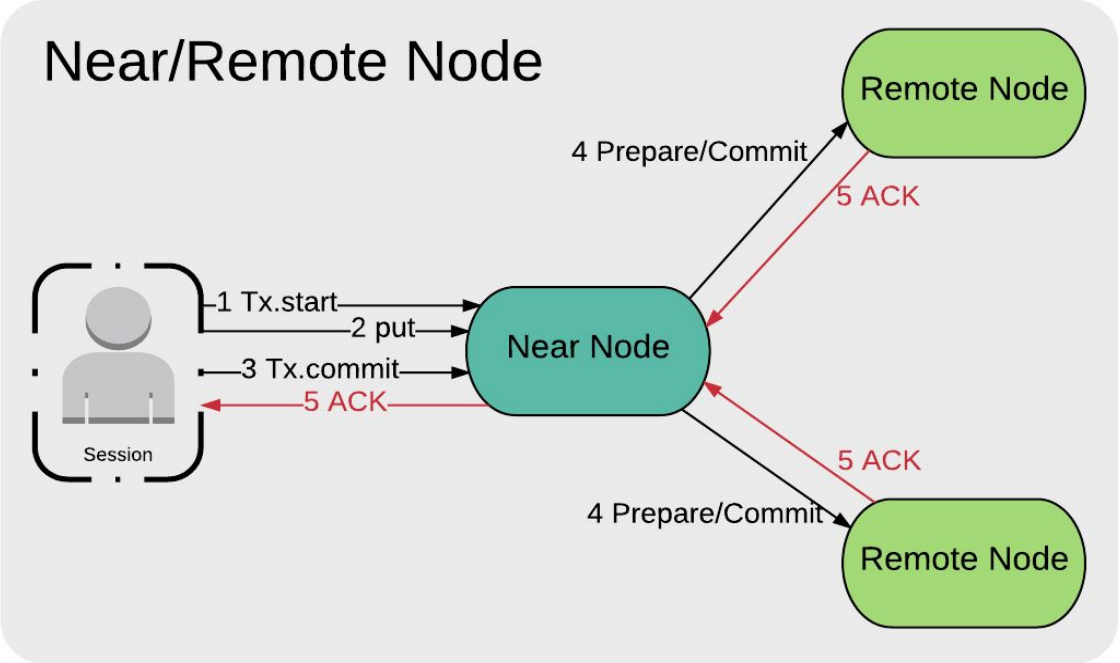


This method ensures that all participating nodes at the same time maintain or celebrate the change transaction.

In the future, we will look at how Ignite comes in cases of "failure" in the different steps of confirmation.

2.2. NearNode / Remote / DHT

If we consider the system in terms of the transaction, it is necessary to introduce another important entity - NearNode.
The algorithm 2 phase commit for each transaction there must be a node which will serve as its "focal point". He will be called NearNode. From the point of view of Ignite - is the node where the transaction was initiated and was called tx.start. Such a node "tracks" transaction state, raised its key, version of the transaction start topology, the nodes involved in the transaction, and other context attributes of the transaction.



In contrast to the Near involved in the transaction partitions are called Remote. They keep "their" part of the cache. Physically, every node allocates at part DHT (Distributed hash table) and through afiniti function, any transaction participant may determine its partition and nodes. It should be noted that DHT keeps the buckets to the level of partitioning, and then use B + tree.

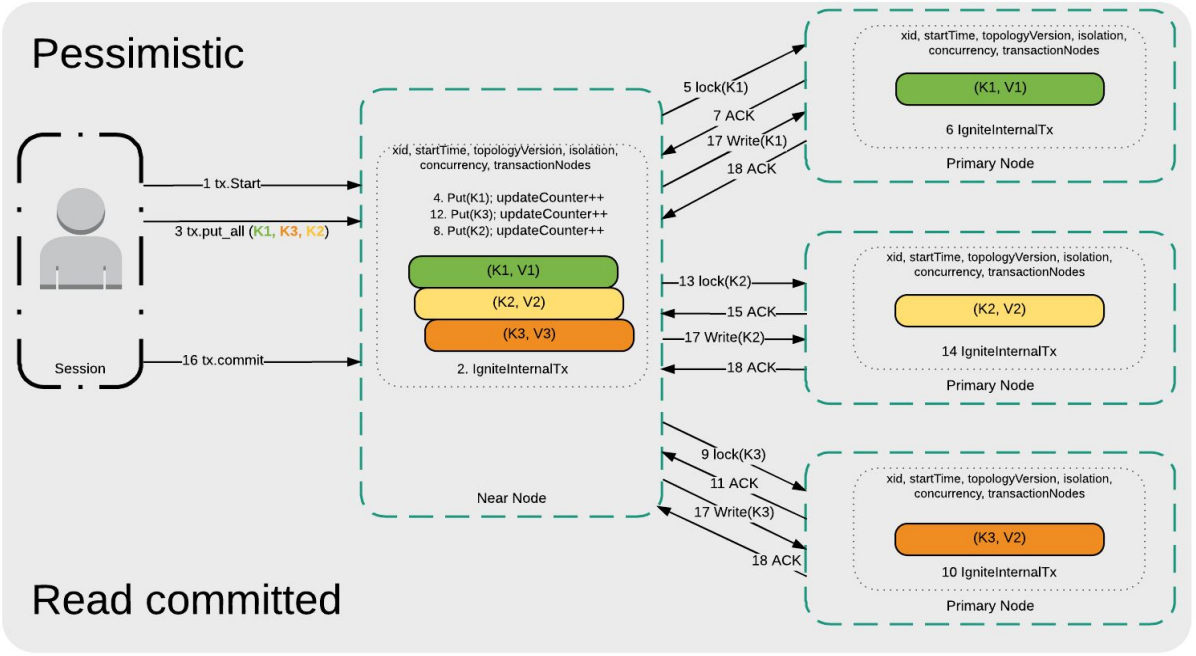
3. Lock mode

3.1. Pessimistic

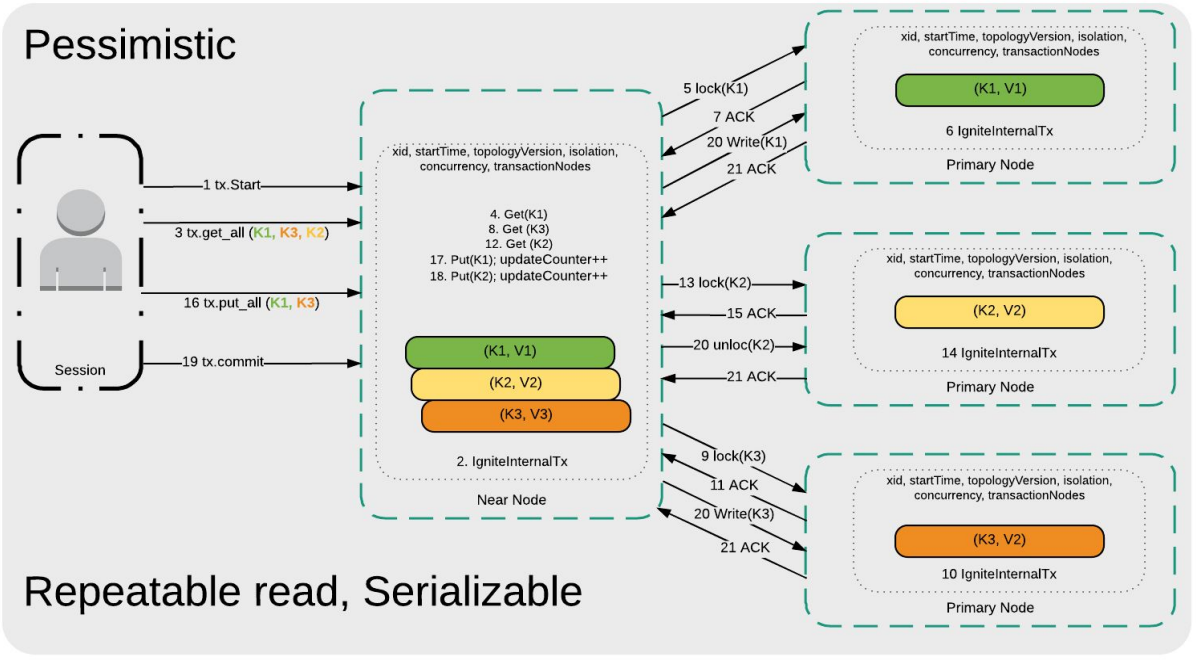
In multi-user systems, different users can simultaneously modify the same data.
To correctly deal with such situations, there are two main approaches - optimistic and pessimistic locking.

When pessimistic locking system first puts a lock on the data, which it plans to change, and then calculates the new values and guaranteed, it modifies data.

Just for the receipt of the lock plays a role, and transaction isolation level.
First, consider the pessimistic regimes.
For Level Read committed block acquired before the recording operation (put, put_all).



The modes and Serializable Repeatable read lock acquired before any operation (read and write).

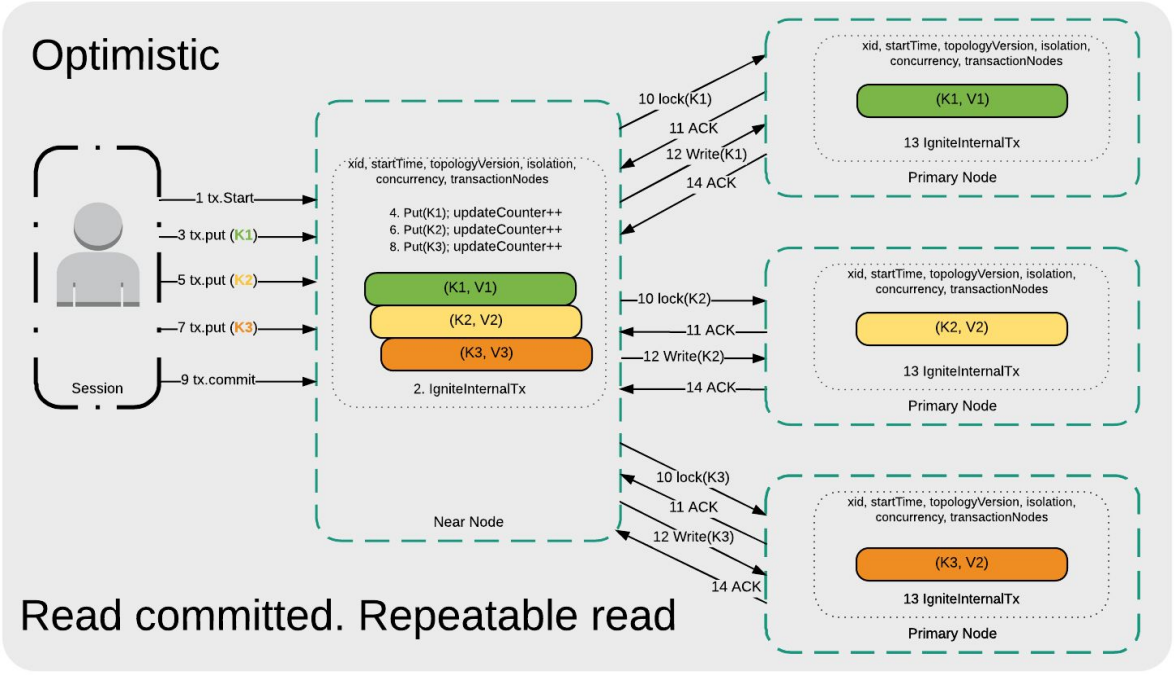


The pessimistic mode data remain locked until the end of the transaction.
To reduce the lock hold time can be used optimistic mode.

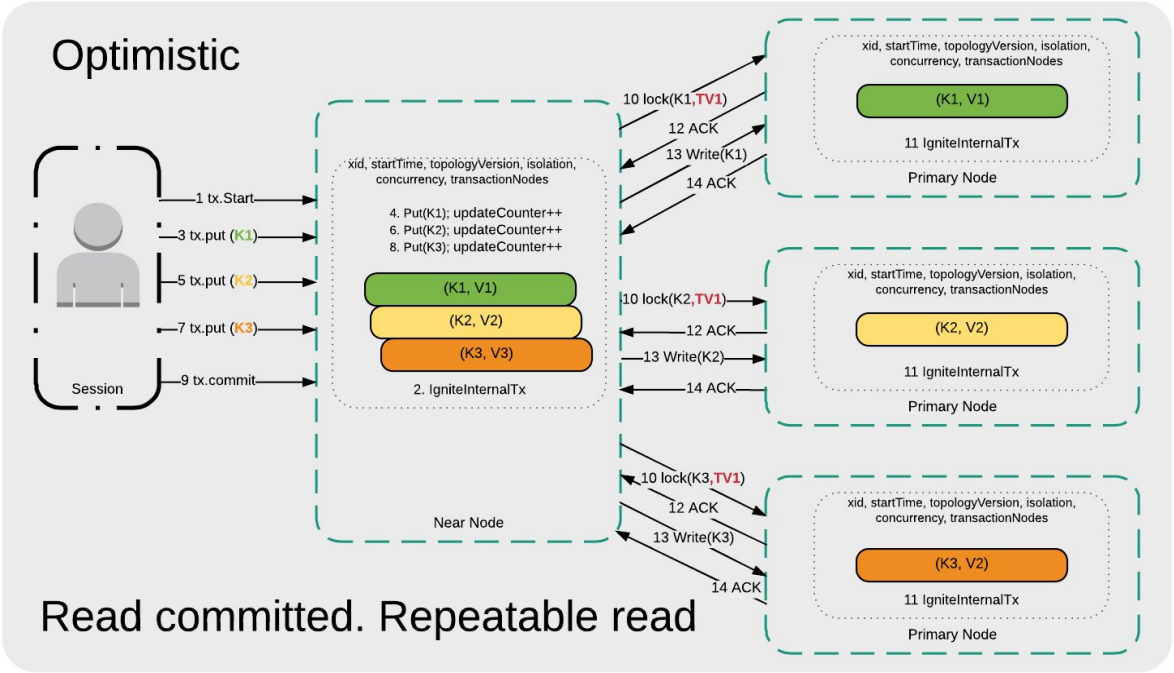
3.2. Optimistic

In this mode, first calculate the new value, and then verify that during the calculation did not happen competing changes. If this is true, the data is locked and modified. If it has been updated, it is necessary to conduct the calculation again.

For the isolation level Read committed and Repeatable read locks are obtained at the time of implementation of phase "prepare", at the same time check that the version has not changed since the beginning of the transaction is not executed.

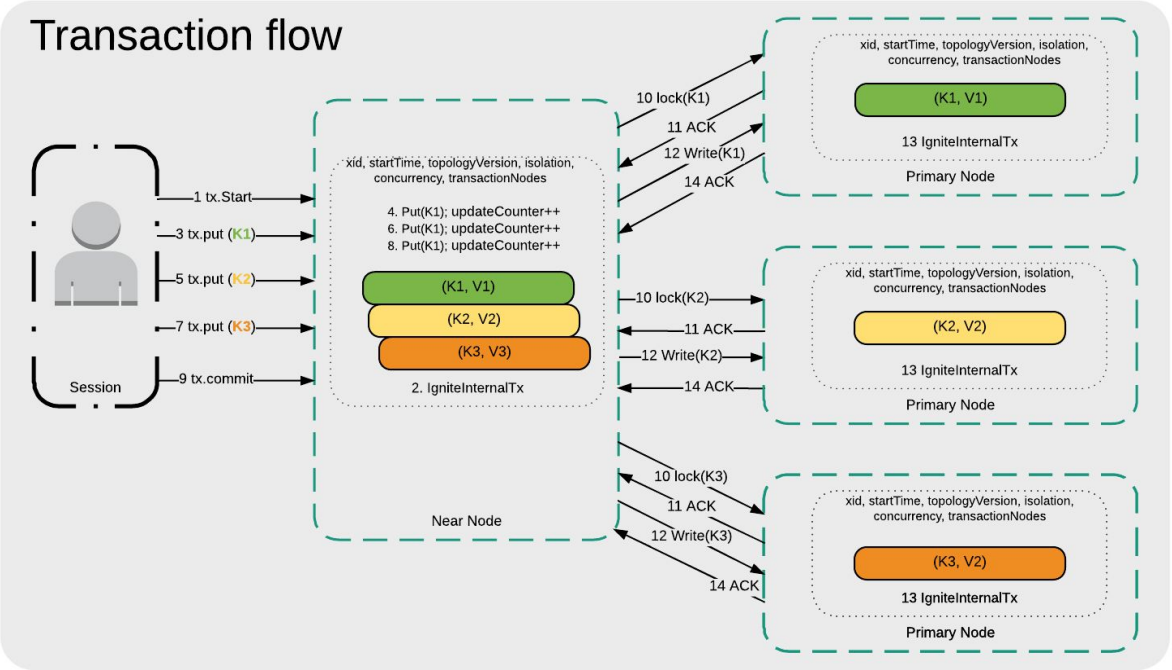


At the level of the Serializable, lock as created in to "prepare" phase, but check the immutability version is performed.



4. Transaction flow

Now let's try to consider the entire life cycle of a transaction in Ignite. For simplicity, assume that the cluster topology is stable, and does not change.



The transaction starts with a start of its operation tx.start. Thus, on the transaction coordinator (NearNode), creates a structure for managing transaction context (interface IgniteInternalTx).

At the same time, for a transactiona

- generating unique transaction identifier
- stored during the transaction began,
- the actual version of the topology,
- the level of isolation and competition for certain transactions, etc.

Transaction Status is set to "active".

Then the algorithm, depending on the level of isolation and competition transaction may differ.
For example, for the transactions with the level of pessimistic, the transaction is immediately checks the opportunity to get a lock on the key, while optimistic, is happening at the moment commit.

All operations performed by the get (in REPEATABLE READ mode) lead to reading data from a cluster and maintain keys and values in the context of a transaction in java heap NearNode.
All operations carried out put also lead to caching of keys and data in the context of, also in java heap NearNode.

TX Rollback

If the user cancels the transaction (performed tx.rollback), then, in pessimistic mode, you must "let go" to get a lock, and delete the transaction context. In the mode "optimistic" lock

acquired only at the time of the commit operation, therefore, simply delete transaction context to NearNode.
There are currently plans to add a mandatory timeout for all transactions. Which will ensure the cancellation, if a transaction exceeds the specified runtime.

TX Commit

If the user commits (tx.commit) is based on the transaction context is created prepare request to perform a step.

At the same time on each primary node sent the information on new or changed keys and, if relevant, the information on how to obtain the lock.

Primary nodes

- check that the version of the transaction topology and host match;
- obtained the required blocking;
- produce DHT context for a transaction and storing the necessary data therein;
- depending on the mode in which the cache is working, or not waiting for confirmation of the success prepare step by Backup nodes;
- inform the coordinator of the transaction's readiness to execute commit.

NearNode sends a commit message, waiting for acknowledgment and translates transaction in COMMITTED status.

Once again it is important to note that all the transaction data should be placed in a heap NearNode memory. Otherwise, the transaction will fail OutOfMemory.

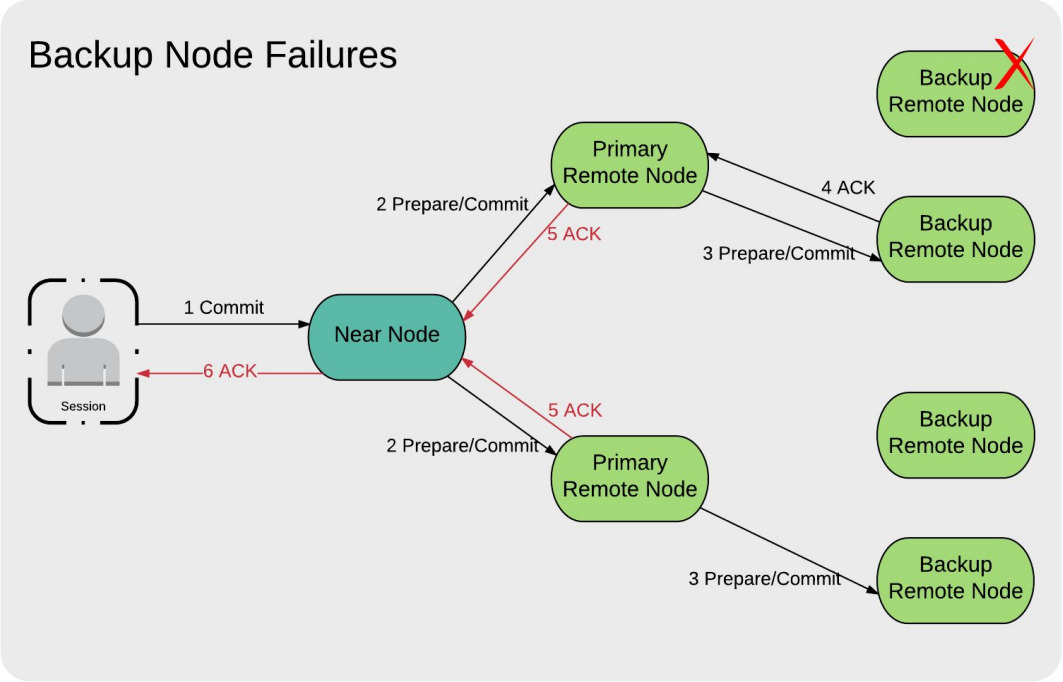
Let us now see how the positive scenario is not processed.

5. Failover / Recovery

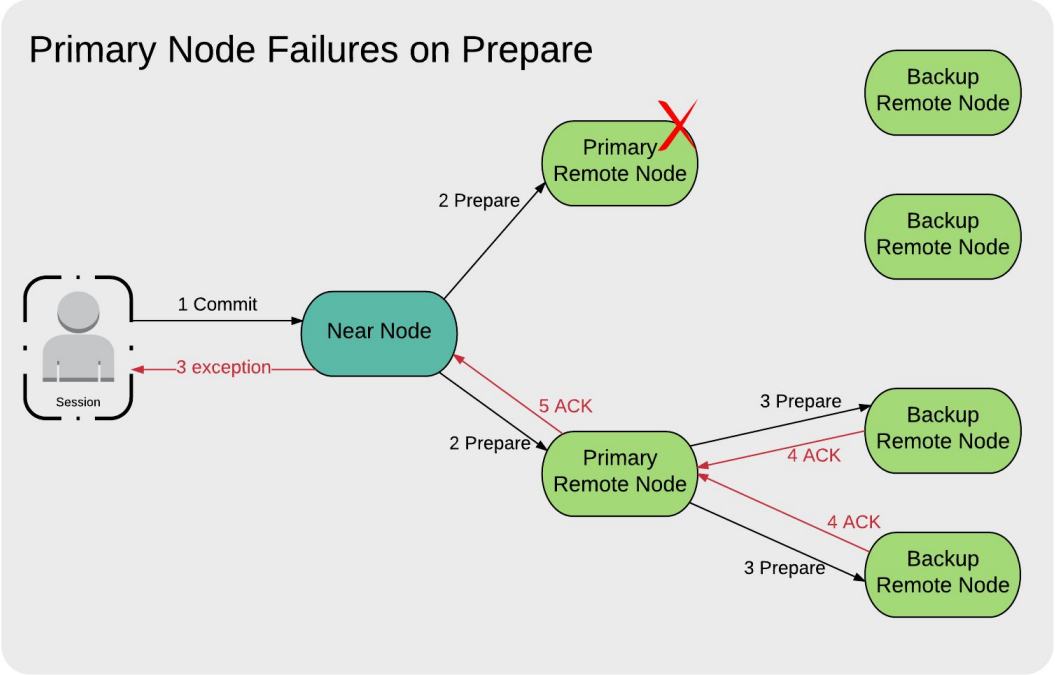
to save the cluster efficiency, especially in systems with a large number of participants, mechanisms are needed to prevent or remedy any failure of the elements.

Backup Node Failures

The simplest case is the failure of a node failure Backup during phase "Prepare" or "Commit". In the case of such an event carry out special action is required. The data is stored on the Primary and Backup of the remaining nodes. Further, outside the Scope transaction, a new backup is created for the partition.



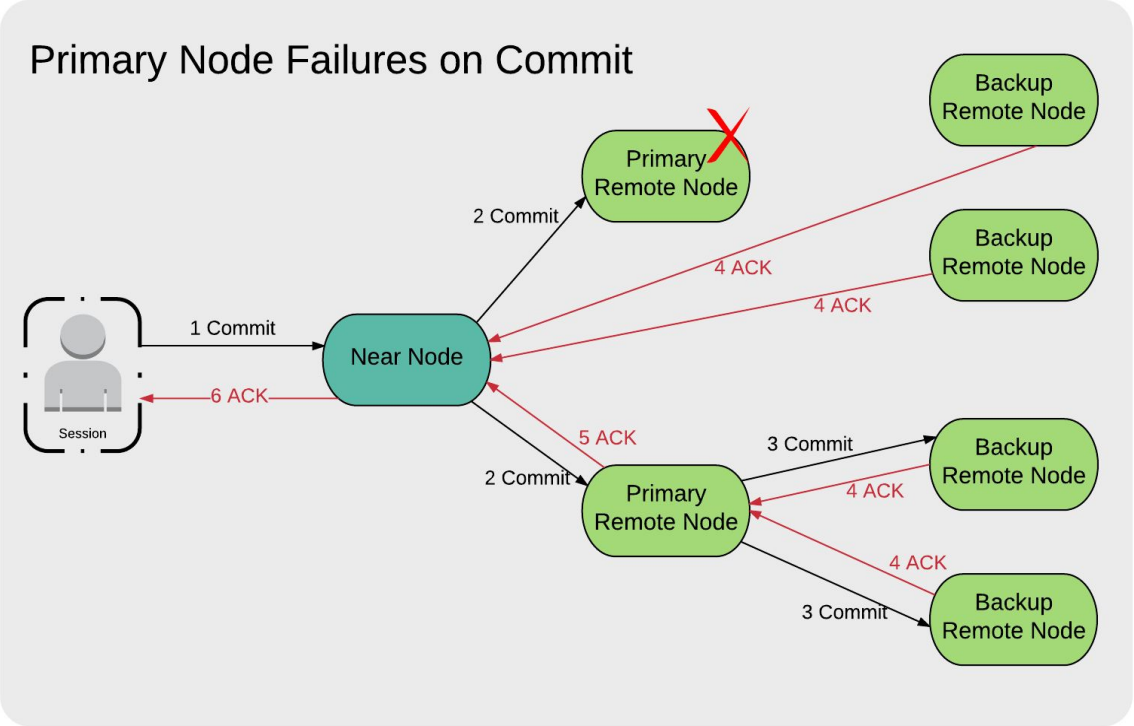
Primary Node Failures on Prepare



To correct the "failure" Primary site required more complex algorithm.
If the failure occurred before or during the "Prepare" stage, and the transaction coordinator was unable to connect to the Primary, it raises an exception, and the client must explicitly take a further decision. By canceling or re-attempt to save the results (but in the new topology, and hence, a new transaction).

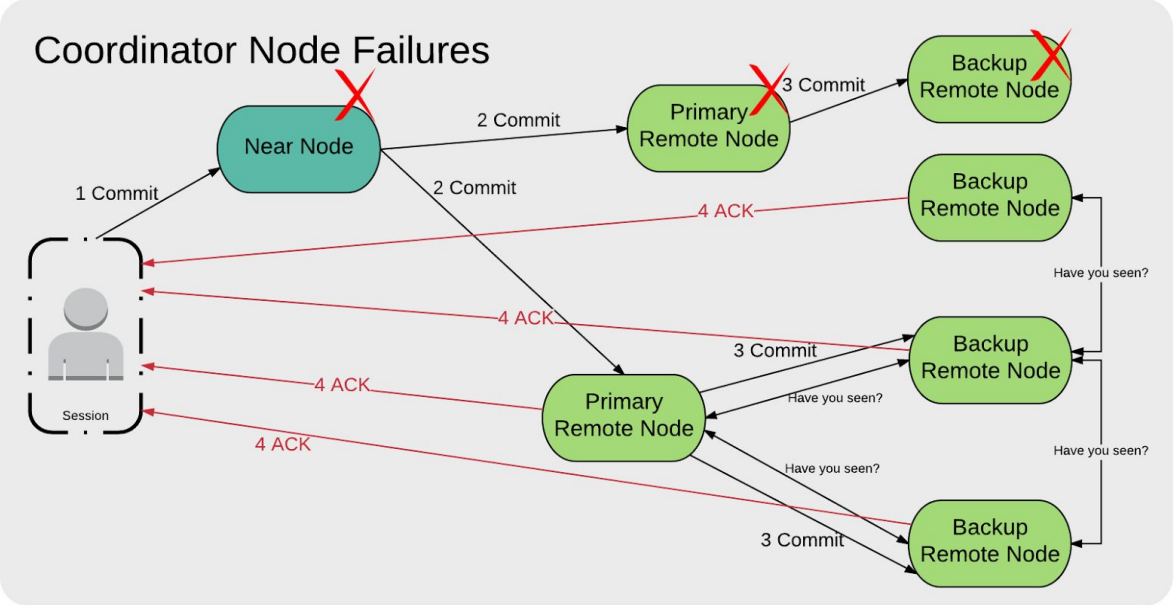
Primary Node Failures on Commit

If failure occurred after "Prepare" stage, the transaction coordinator, determining unavailability Primary node will wait (NodeFailureDetection) response from the Backup nodes.
Backup nodes, identifying "missing" the Primary, sending the message to the coordinator of the transaction (the initiator of the information, and all participants in the transaction is included in the message received at the prepare phase).
The topology of the cluster is not changed since you need to change the passage of PME message, and it will wait for the active transactions.
Transaction coordinator confirms it, the transaction is not active, the topology is changed.



Coordinator Node Failures

When "refusal" Coordinator transactions require more action. Each participant has the information only on the local status of the transaction and, most of all, this status will be different on different nodes. It is possible that some of the participants had already received a "Commit" message, or someone has not received "Prepare". Therefore, all the Primary nodes send messages to the current status information. If one of the nodes will respond that the message "Prepare" have been reported to it (or do not respond at all), then the transaction is aborted on all nodes.


Coordinator Node Failures
Near Node
Primary Remote Node
Backup Remote Node
Backup Remote Node
Backup Remote Node
Backup Remote Node
Primary Remote Node
Backup Remote Node
Session
1 Commit
2 Commit
2 Commit
3 Commit
4 ACK
4 ACK
4 ACK
4 ACK
3 Commit
Have you seen?
Have you seen?
Have you seen?
Have you seen?
3 Commit