w: www.mariadb.com

e: info@mariadb.com

MySQL Cluster Internal Architecture

Max Mether
MariaDB

Keywords:
MySQL Cluster, MySQL, High Availability, HA, Clustering, Architecture

1 Introduction

MySQL Cluster is tightly linked to the MySQL server, yet it is a separate product. Due to its distributed nature, MySQL Cluster has a far more complicated architecture than a standard MySQL database. This presentation and white paper will describe the internal architecture of the MySQL Cluster product and the different mechanisms and process that exists to insure the availability of the cluster. We will also take a close look at the data and message flow that takes place between the nodes in the cluster during different types of operations.

2 MySQL Cluster Features

MySQL Cluster can be distinguished from many other clustering products in that it offers five nines availability with a shared-nothing architecture. Most clustering products use a shared-disk architecture. Using a shared-nothing architecture allows the cluster to run on commodity hardware, thus greatly reducing costs compared to shared-disk architecture. Having a shared-nothing architecture does, however, come with a different cost, namely, a more complex way of ensuring high availability and synchronization of the nodes which is mainly seen as increased network traffic.

MySQL Cluster also provides ACID transactions with row level locking using the READ-COMMITTED isolation level. MySQL Cluster is, in general, an in-memory database, however non-indexed data can be stored on disk and the disks are also used for checkpointing to ensure durability through system shutdown. MySQL cluster provides unique hash indexes as well as ordered T-tree indexes. It also supports several online features, including a native online backup, and the ability perform certain ALTER TABLE operations with the tables unlocked (most storage engines require table locks for all ALTER TABLE operations)

3 Architecture

The MySQL Cluster architecture can be divided into several layers. The first is the application layer where the applications that communicate with the MySQL servers reside. These applications are normal MySQL clients from a MySQL server perspective and all communication with the lower layers of the cluster is handled by the MySQL servers.

The second layer is the SQL layer where the MySQL servers reside, called SQL or API nodes in the cluster. The number of MySQL servers is completely independent of the number of data nodes, which allows great flexibility in a cluster's configuration. In particular as the number of SQL nodes can be increased without shutting down the cluster. In addition to MySQL servers, other programs that communicate directly with the data nodes (such as restoration programs or programs to view the clustered tables) are also considered API nodes. MySQL Cluster

offers a C API facilitate the creation of programs that communicate directly with the data nodes without passing through a MySQL server. All such programs would be considered API nodes.

Next is the data layer where the data nodes reside. They manage all the data, indexes and transactions in the cluster and are thus responsible for the availability and the durability of the data.

As well, there is a management layer where the management server(s) reside.
The rest of this paper/presentation will focus on the data nodes.

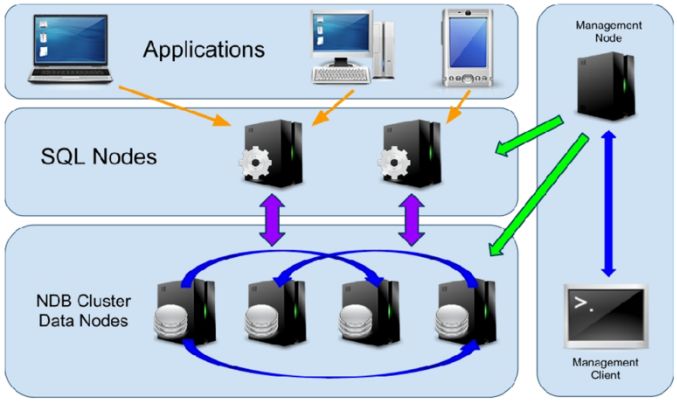

Illustration. 1: The MySQL Cluster Architecture

4 Partitioning

Let's take a closer look at the data managed by the data nodes. Each table in the cluster is partitioned based on the number of data nodes. This means that if the cluster has two data nodes, each table is partitioned in two parts, and if the cluster has four data nodes, each table is partitioned into four parts. The partitioning is done based on the hash value of the primary key, by default. The partitioning function can be changed if needed, but for most cases the default is good enough. Each data node holds the so called **primary** replica or fragment for a partition, allowing the data to be distributed evenly between all the data nodes. To ensure the availability (and redundancy) of the data, each node also holds a copy of another partition, called a **secondary** replica. The nodes work in pairs so that the node holding the secondary partition of another node's primary partition, will reciprocate and give its own primary partition as a secondary partition to the same node partner. These pairs are called node groups and there will be #nodes / 2 node groups in the cluster. This means that for a cluster with 2 data nodes, each node will contain the whole database, but for a cluster with 4 nodes, each node will only contain half of the data.

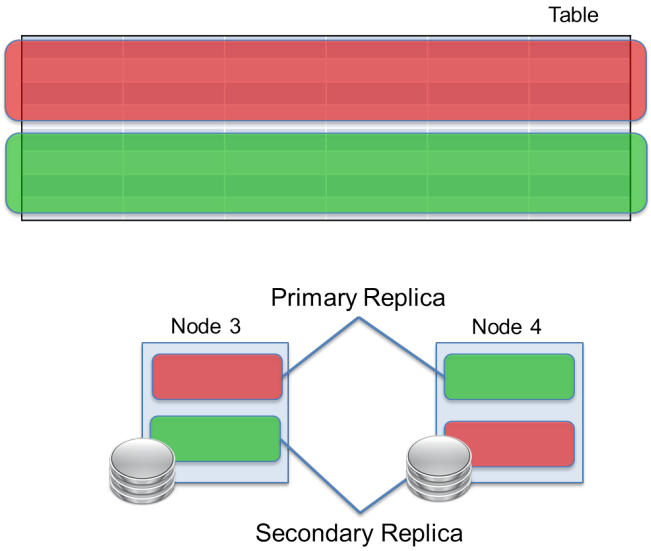

Illustration. 2: Data partitioning with 2 data nodes

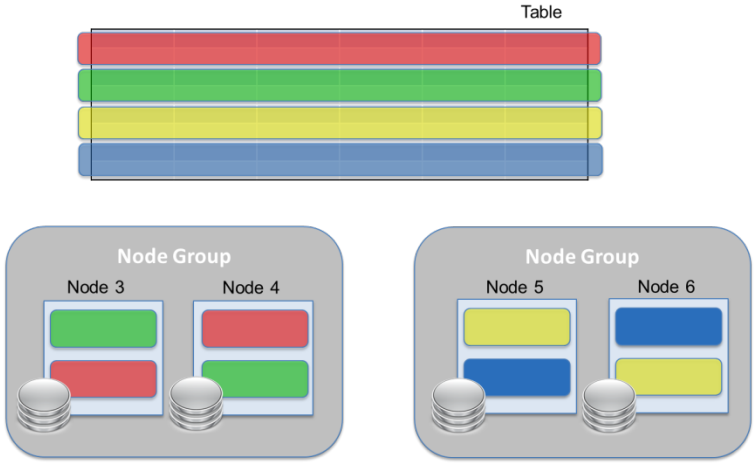

Illustration. 3: Data partitioning with 4 data nodes

5 Heartbeat circle

Because of the shared-nothing architecture, all nodes in the cluster must always have the same view of who is connected to the cluster. Detecting failed nodes is extremely important. In MySQL Cluster this can either be handled through TCP close or through the heartbeat circle.

The data nodes are organised in a logical circle where each node sends heartbeats to the next node in the circle. If a node fails to send 3 consecutive heartbeats, the following node assumes that the former node has crashed or cannot communicate, and launches the network partitioning protocol. While this is occurring, all processing in the cluster is temporarily stopped and the remaining nodes determine whether they can continue or not. The node that is not responding is excluded from the cluster and the remaining nodes form a “new” cluster without the unresponsive node. Note that in this new cluster, the partner of the unresponsive node will now contain two primary fragments, as the fragment that was previously a secondary replica is now promoted to primary status. This also means that this node will handle twice the amount of traffic as before the crash. All nodes must thus be tuned to be able to handle twice the normal workload.

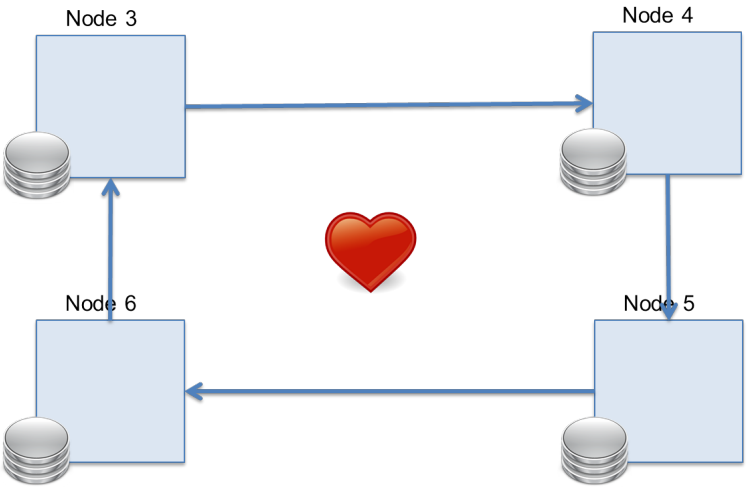

Illustration. 4: Heartbeat circle between 4 data nodes

6 Network Partitioning Protocol

The network partitioning protocol is fundamental to guaranteeing the availability of the cluster. The largest problem derives from the fact that a network failure is indistinguishable from a node crash, from another node's perspective. This means that precautions must be in place so that a split-brain scenario cannot occur when the cause of the communication failure is a network failure and not an actual node crash. When there is a network partitioning, the network partitioning protocol is launched on both sides of the split and it guarantees that, in the event of an even split, there will only be one running "cluster" remaining. The protocol to be followed by each set of nodes that can communicate with each other is the following:

1. Do we have at least one node from each node group? If no, then we shutdown,
2. Do we have all nodes from any of the node groups? If yes, we can continue as the cluster.
3. We ask the arbitrator to decide our fate.

The arbitrator follows a simple rule: the first set of nodes to ask will be given a positive answer, and all other sets will be given a negative answer. If a set of nodes cannot contact the arbitrator, they will shut down automatically.

7 Durability

Since the MySQL Cluster is generally an in-memory database some measures have to be taken so that a cluster shutdown is possible without incurring data loss. The data nodes use two procedures to maintain data on disk. The first is the REDO log. When a transaction takes place on a node, the transaction is stored in a REDO log buffer which is synchronously flushed to disk at even intervals. The disk based REDO log will only contain the transactions that were committed at the point in time the flush took place. The flushing takes place through a global checkpoint or GCP.

Of course, such a REDO log would continue to grow ad infinitum, so the process needs to be limited. This is done through another process called a local checkpoint or LCP. During an LCP, the data nodes store a snapshot all their data on disk allowing all REDO log contents before this point in time to be discarded. For safety reasons the data nodes keep two LCPs on disk and thus need REDO logs from the time of the start of the first LCP through to

the beginning of the third LCP, for a recovery to be successful. The LCPs are written in a circular fashion (LCP 3 will replace LCP 1 etc), as is the REDO log.

The contents of the LCPs and the REDO log are used when a node recovers- it can rebuild data up to its last GCP from its own disk and then transfer the remaining changes from its partner node before regaining primary status for any of its fragments. When the whole cluster is shutdown gracefully, a last GCP is issued before shutting down the data nodes. Each data node can then recover its data completely from its own disk.

8 Transactions

MySQL Cluster provides synchronous replication between the data nodes, which means that the replication is synchronous from the client's perspective. In order to achieve this, a complex algorithm is used called the two-phase commit protocol (2PC). In the 2PC, a transaction is committed through two phases: a prepare phase and a commit phase. During the prepare phase, the nodes perform the requested operations and are ready to commit the transaction. During the commit phase, the transaction is committed and the changes cannot be undone anymore.

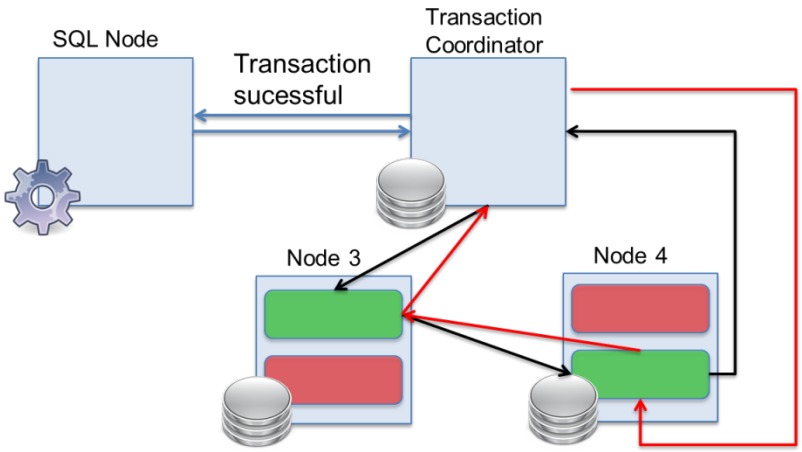

Illustration. 5: Two-phase commit protocol

Let's look at the actual implementation of this in MySQL Cluster using a simple primary key write as an example. First the MySQL server (or whichever SQL node the request comes from) contacts a data node. Every data node has a transaction coordinator (TC) module and the contacted data node now becomes the coordinator, or manager, for that transaction. Given the primary key value, the TC calculates in which partition the row resides and forwards the operation to the data node holding the primary replica of that partition. On the primary node, the operation is performed and the required row is locked, where upon the same operation is sent to the partner node holding the secondary replica. Once the secondary node performs the operation, the TC is contacted again and the operation can now be committed.

The commit phase follows the same sequence but in the reverse order; the node with the secondary replica is contacted first and then the node with the primary replica. Once the primary node has committed the operation, the TC acknowledges the operation as committed to the client.
All in all, it's a total of 6 internal messages sent between nodes, and 2 messages between the client and the TC, for this simple primary key operation. If a transaction contains multiple operations, the internal steps are repeated for each operation. Operations other than primary key writes follow somewhat different work flows but the principle remains the same.

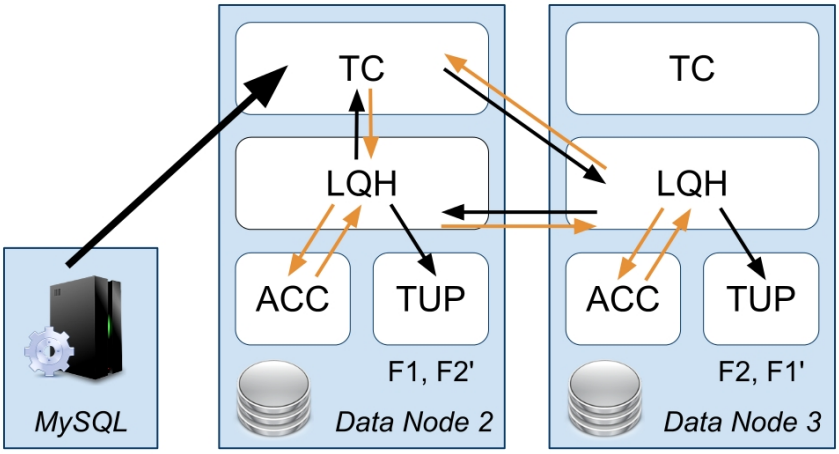

Illustration. 5: A primary key write in a cluster with 2 data nodes