**Capitolul 2. Conceptul de MariaDB Cluster**

MariaDB se folosește de Galera Cluster pentru implementarea clusterului. Acesta va fi discutat în detaliu la punctul 2.2. În continuare vom afla ce reprezinta un cluster și care sunt arhitecturile acestuia.

**2.1. Conceptul de clustering**

Users often implement databases in a clustered configuration in order to accommodate business requirements such as scalability, performance, high-availability and failure recovery. When implementing a cluster, the architecture of the underlying DBMS is an important factor to consider. Cluster computing is “a group of linked computers, working together closely so that in many respects they form a single computer.” In the database vernacular, clustering means that the application sees a single database, while under the covers there are two or more computers processing the data. In addition to providing scalability, database clusters can deliver additional benefits such as load balancing and high-availability, but these things are not inherent in all database clusters.

**2.2. Arhitecturile unui cluster**

The two predominant DBMS architectures are **shared-nothing** and **shared-disk**.

**2.2.1. Shared-Nothing**

Shared-nothing works on the principle that each node in a cluster has sole ownership of the data on that node. Each node literally shares no data with the other nodes of the cluster, hence the term shared-nothing. When you move from a single server to multiple servers in a shared-nothing cluster, you must divide the data across the servers. This process of splitting the data across servers, as indicated in the diagram below, is called partitioning. Data can be partitioned vertically or horizontally.

Figura 2.1 Shared-Nothing (partiționarea datelor între noduri)

Requests for data are then processed through a routing table that routes each request to the server/node that owns that data. For example, the Server 1 above may have information about users, while Server 2 might have information about orders. If your application makes a request that involves both servers, for example requesting a list of users and order information for orders placed in the prior month, you need to involve both servers. The database would solve this request by reading the list of orders placed the prior month and then sending that list from Server 2 to Server 1 to add the information about the users. This process of sending data from one server to another is called data shipping. At its core, a shared-nothing database is a standalone database with the added facility for data shipping.

**Fail-Over and the Master-Slave Configuration**: If a server fails, all applications that require access to the failed server also fail. For this reason, each node needs a mirrored copy that can take over in case the primary node fails. This node is called a slave. In the master-slave configuration, the master can perform both reads and updates, while the slave can only provide read access to the slave’s local copy of the data

Figura 2.2 Configurație Master-Slave

For applications with a high ratio of reads versus updates, the slaves can be configures in a tree, where the data from the master is replicated down the tree, and the slaves then off-load the readaccess.

Figura 2.3 Arbore de Slave

There is nothing to prevent a shared-disk from using a similar tree of read-only slaves, but slaves are generally associated with shared-nothing.

Examples of Shared-Nothing DBMS: Oracle 11g, IBM DB/2 (non-mainframe), Sybase ASE, MySQL (InnoBD, MyISAM, Falcon…all storage engines except ScaleDB), Microsoft SQL Server, etc.

**2.2.2. Shared-Disk**

Shared-disk, also known as shared-everything, works on the principle that you have one array of disks , typically a Storage Area Network (SAN), or Network Attached Storage (NAS), that holds all of the data in that database. Each server or node in the cluster acts on that single collection of data in real-time.

Figura 2.4 Cluster Shared-Disk (toate nodurile au acces la toate datele)

In a shared-disk architecture, any node can satisfy any request, because they all have access to all of the data. So, instead of going to a specific node for specific data, shared-disk can simply route the request to the next available node. Because each node can address any database request, the load is inherently balanced across the nodes in the cluster.

**Fail-Over, Load Balancing and the Master-Master Configuration**: While shared-nothing has only one node that can update any piece of data in the database, shareddisk enables any node to update the database. For this reason, shared-disk is called a mastermaster architecture. As such, it fully utilizes each server in the cluster. This architecture also provides inherent failover, since each node acts as a back-up to every other node. Finally, the shared-disk architecture provides inherent load-balancing, since each database request can be sent to any available server.

However, nothing comes for free. In order to enable this flexibility, the nodes communicate with each other to coordinate their activity, specifically: locking, buffering and status. This inter-nodal messaging creates some degree of overhead, but you must balance this overhead against the impact of data/function shipping found in shared-nothing clusters.

**Examples of Shared-Disk DBMS**: ScaleDB (MySQL), Oracle RAC, Sybase ASE CE, IBM IMS and DB/2 (mainframe). ScaleDB in the only shared-disk solution for MySQL.

**Which Architecture is Better?** Unfortunately, such a simple question does not have a simple answer. Each architecture has its pros and cons. In many ways, comparing the two architectures is like comparing automobile transmissions. Shared-disk is analogous to the automatic transmission, because it automates much of the complexity of set-up and maintenance, thus lowering total cost of ownership (TCO). Shared-nothing is analogous to the manual transmission because it provides more granular control in exchange for increased manual effort on the part of the owner. And just as drivers tend to be passionate about their preferred transmission, DBAs tend to be passionate about their preferred database architecture.

**2.3 MariaDB Galera**

MariaDBGalera Cluster is a synchronous multi-master database cluster, based on synchronous replication and Oracle’s MySQL/InnoDB. When Galera Cluster is in use, you can direct reads and writes to any node, and you can lose any individual node without interruption in operations and without the need to handle complex failover procedures.

At a high level, Galera Cluster consists of a database server—that is, MySQL, MariaDB or Percona XtraDB—that then uses the Galera Replication Plugin to manage replication. To be more specific, the MySQL replication plugin API has been extended to provide all the information and hooks required for true multi-master, synchronous replication. This extended API is called the Write-Set Replication API, or wsrep API.

Through the wsrep API, Galera Cluster provides certification-based replication. A transaction for replication, the write-set, not only contains the database rows to replicate, but also includes information on all the locks that were held by the database during the transaction. Each node then certifies the replicated write-set against other write-sets in the applier queue. The write-set is then applied, if there are no conflicting locks. At this point, the transaction is considered committed, after which each node continues to apply it to the tablespace.

This approach is also called virtually synchronous replication, given that while it is logically synchronous, the actual writing and committing to the tablespace happens independently, and thus asynchronously on each node.

**2.3.1. Avantaje**

Galera Cluster provides a significant improvement in high-availability for the MySQL ecosystem. The various ways to achieve high-availability have typically provided only some of the features available through Galera Cluster, making the choice of a high-availability solution an exercise in tradeoffs.

The following features are available through Galera Cluster:

* True Multi-master Read and write to any node at any time.
* Synchronous Replication No slave lag, no data is lost at node crash.
* Tightly Coupled All nodes hold the same state. No diverged data between nodes allowed.
* Multi-threaded Slave For better performance. For any workload.
* No Master-Slave Failover Operations or Use of VIP.
* Hot Standby No downtime during failover (since there is no failover).
* Automatic Node Provisioning No need to manually back up the database and copy it to the new node.
* Supports InnoDB.
* Transparent to Applications Required no (or minimal) changes) to the application.
* No Read and Write Splitting Needed.

The result is a high-availability solution that is both robust in terms of data integrity and high-performance with instant failovers.

**Cloud Implementations with Galera Cluster**

An additional benefit of Galera Cluster is good cloud support. Automatic node provisioning makes elastic scale-out and scale-in operations painless. Galera Cluster has been proven to perform extremely well in the cloud, such as when using multiple small node instances, across multiple data centers—AWS zones, for example—or even over Wider Area Networks.

**2.3.2. Arhitecura**

**REPLICATION API**

Synchronous replication systems use eager replication. Nodes in the cluster synchronize with all other nodes by updating the replicas through a single transaction. Meaning that, when a transaction commits, all nodes have the same value. This process takes place using write-set replication through group communication.

The internal architecture of Galera Cluster revolves around four components:

* **Database Management System (DBMS)** The database server that runs on the individual node. Galera Cluster can use MySQL, MariaDB or Percona XtraDB.
* **wsrep API** The interface and the responsibilities for the database server and replication provider. It consists of:

**- *wsrep hooks*** The integration with the database server engine for write-set replication.

**- *dlopen()*** The function that makes the wsrep provider available to the wsrep hooks.

* **Galera Replication Plugin** The plugin that enables write-set replication service functionality.
* **Group Communication plugins** The various group communication systems available to Galera Cluster. For instance, gcomm and Spread.

**WSREP API**

The wsrep API is a generic replication plugin interface for databases. It defines a set of application callbacks and replication plugin calls.

The wsrep API uses a replication model that considers the database server to have a state. The state refers to the contents of the database. When a database is in use, clients modify the database content, thus changing its state. The wsrep API represents the changes in the database state as a series of atomic changes, or transactions.

In a database cluster, all nodes always have the same state. They synchronize with each other by replicating and applying state changes in the same serial order.

From a more technical perspective, Galera Cluster handles state changes in the following process:

1. On one node in the cluster, a state change occurs on the database.
2. In the database, the wsrep hooks translate the changes to the write-set.
3. dlopen() makes the wsrep provider functions available to the wsrep hooks.
4. The Galera Replication plugin handles write-set certification and replication to the cluster.

For each node in the cluster, the application process occurs by high-priority transaction(s).

**Global Transaction ID**

In order to keep the state identical across the cluster, the wsrep API uses a Global Transaction ID, or GTID. This allows it to identify state changes and to identify the current state in relation to the last state change.

45eec521-2f34-11e0-0800-2a36050b826b:94530586304

The Global Transaction ID consists of the following components:

* **State UUID** A unique identifier for the state and the sequence of changes it undergoes.
* **Ordinal Sequence Number** The seqno, a 64-bit signed integer used to denote the position of the change in the sequence.

The Global Transaction ID allows you to compare the application state and establish the order of state changes. You can use it to determine whether or not a change was applied and whether the change is applicable at all to a given state.

**Galera Replication Plugin**

The Galera Replication Plugin implements the wsrep API. It operates as the wsrep Provider. From a more technical perspective, the Galera Replication Plugin consists of the following components:

* **Certification Layer** This layer prepares the write-sets and performs the certification checks on them, ensuring that they can be applied.
* **Replication Layer** This layer manages the replication protocol and provides the total ordering capability.
* **Group Communication Framework** This layer provides a plugin architecture for the various group communication systems that connect to Galera Cluster.

**Group Communication Plugins**

The Group Communication Framework provides a plugin architecture for the various gcomm systems.

Galera Cluster is built on top of a proprietary group communication system layer, which implements a virtual synchrony QoS. Virtual synchrony unifies the data delivery and cluster membership services, providing clear formalism for message delivery semantics.

While virtual synchrony guarantees consistency, it does not guarantee temporal synchrony, which is necessary for smooth multi-master operations. To get around this, Galera Cluster implements its own runtime-configurable temporal flow control. Flow control keeps nodes synchronized to the faction of a second.

In addition to this, the Group Communication Framework also provides a total ordering of messages from multiple sources. It uses this to generate Global Transaction ID‘s in a multi-master cluster.

At the transport level, Galera Cluster is a symmetric undirected graph. All database nodes connect to each other over a TCP connection. By default TCP is used for both message replication and the cluster membership services, but you can also use UDP multicast for replication in a LAN.

**2.3.3. Nivele de izolare**

Galera Cluster handles transactions in isolation. These isolation levels guarantee that the nodes process transactions in a reliable manner.

Isolation ensures that concurrently running transactions do not interfere with each other. Because of this, it also ensures data consistency. If the transactions were not isolated, one transaction could modify data that other transactions are reading, which would lead to data inconsistency.

Galera Cluster employs four isolation levels, which are in ascending order:

* READ-UNCOMMITTED
* READ-COMMITED
* REPEATABLE-READ
* SERIALIZABLE

**READ-UNCOMMITTED**

Here transactions can see changes to data made by other transactions that are not yet committed. In other words, transactions can read data that eventually may not exist, given that other transactions can always rollback the changes without commit. This is known as a dirty read. Effectively, READ-UNCOMMITTED has no real isolation at all.

**READ-COMMITTED**

Here dirty reads are not possible. Uncommitted changes remain invisible to other transactions until the transaction commits. However, at this isolation level SELECT queries use their own snapshots of committed data, that is data committed before the SELECT query executed. As a result, SELECT queries, when run multiple times within the same transaction, can return different result sets. This is called a non-repeatable read.

**REPEATABLE-READ**

Here non-repeatable reads are not possible. Snapshots taken for the SELECT query are taken the first time the SELECT query runs during the transaction. The snapshot remains in use throughout the entire transaction for the SELECT query. It always returns the same result set. This level does not take into account changes to data made by other transactions, regardless of whether or not they have been committed. IN this way, reads remain repeatable.

**SERIALIZABLE**

Here all records accessed within a transaction are locked. The resource locks in a way that also prevents you from appending records to the table the transaction operates upon. SERIALIZABLE prevents a phenomenon known as a phantom read. Phantom reads occur when, within a transaction, two identical queries execute, and the rows the second query returns differ from the first.

Galera Cluster uses transaction isolation at both the local and the cluster level.

**Local Transaction Isolation**

Transaction isolation occurs on each node at the local level of the database server. It functions the same as with the native InnoDB storage engine. All four levels are available. The default setting for local transaction isolation is REPEATABLE-READ.

**Cluster Transaction Isolation**

Transaction isolation also occurs at the cluster level. Between transactions processing on separate nodes, Galera Cluster implements a transaction level called SNAPSHOT-ISOLATION. TheSNAPSHOT-ISOLATION level occurs between REPEATABLE-READ and SERIALIZABLE.

The reason for this is that there is no support in the SERIALIZABLE transaction isolation level for the multi-master use case, neither in the STATEMENT nor the ROW formats. This is due to the fact that the Galera Replication Plugin does not carry a transaction read-set. Also, because the SERIALIZABLEtransaction isolation level is vulnerable to multi-master conflicts. It holds read locks and any replicated writes to a read locked row cause the transaction to abort.

It is recommended that you avoid using SERIALIZABLE in Galera Cluster.

**2.4. Funcționarea unui cluster MariaDB Galera**

The primary focus is data consistency. The transactions are either applied on every node or not all. So, the databases stay synchronized, provided that they were properly configured and synchronized at the beginning.

The Galera Replication Plugin differs from the standard MariaDB Replication by addressing several issues, including multi-master write conflicts, replication lag and slaves being out of sync with the master.

Figura 2.5 Funcționarea unui cluster MariaDB Galera

In a typical instance of a Galera Cluster, applications can write to any node in the cluster and transaction commits, are then applied to all the servers, through certification-based replication.

Certification-based replication is an alternative approach to synchronous database replication, using group communication and transaction ordering techniques.

Configurarea unui cluster MariaDB Galera este prezentată în capitolul 4, în descrierea aplicației.

**2.5. Limitări**

Vom enumera câteva limitări importante ale clusterului MariaDB Galera:

* Windows is not supported
* Currently replication works only with the InnoDB storage engine. Any writes to tables of other types, including system (mysql.\*) tables are not replicated (this limitation excludes DDL statements such as CREATE USER, which implicitly modify the mysql.\* tables — those are replicated). There is however experimental support for MyISAM - see the wsrep\_replicate\_myisam system variable)
* Unsupported explicit locking include LOCK TABLES, FLUSH TABLES {explicit table list} WITH READ LOCK, (GET\_LOCK(), RELEASE\_LOCK(),…). Using transactions properly should be able to overcome these limitations. Global locking operators like FLUSH TABLES WITH READ LOCK are supported.
* All tables should have a primary key (multi-column primary keys are supported). DELETE operations are unsupported on tables without a primary key. Also, rows in tables without a primary key may appear in a different order on different nodes.
* The query log cannot be directed to a table. If you enable query logging, you must forward the log to a file:log\_output=FILE
* XA transactions are not supported.
* Transaction size. While Galera does not explicitly limit the transaction size, a writeset is processed as a single memory-resident buffer and as a result, extremely large transactions (e.g. LOAD DATA) may adversely affect node performance. To avoid that, the wsrep\_max\_ws\_rows and wsrep\_max\_ws\_size system variables limit transaction rows to 128K and the transaction size to 1Gb by default. If necessary, users may want to increase those limits. Future versions will add support for transaction fragmentation.