MySQL™ Group Replication

the Magic Explained

Frédéric Descamps

OPEN SOURCE DATABASE CONFERENCE

PERCONA LIVE EUROPE FRANKFURT


ORACLE®

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purpose only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied up in making purchasing decisions. The development, release and timing of any features or functionality described for Oracle´s product remains at the sole discretion of Oracle.

about.me/lefred

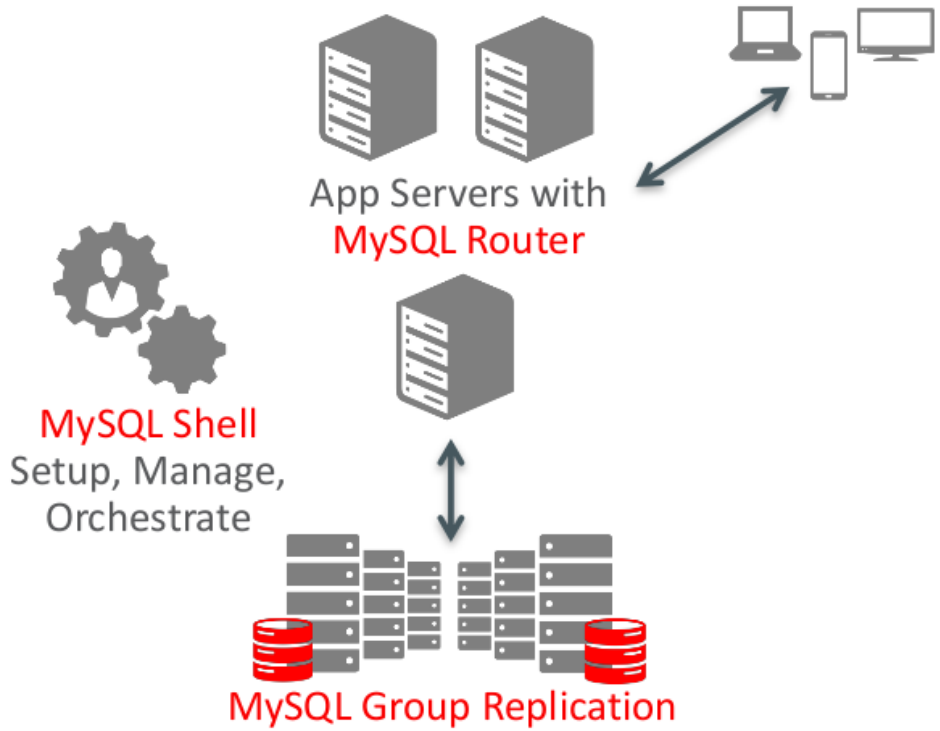
Who am I ?

Frédéric Descamps

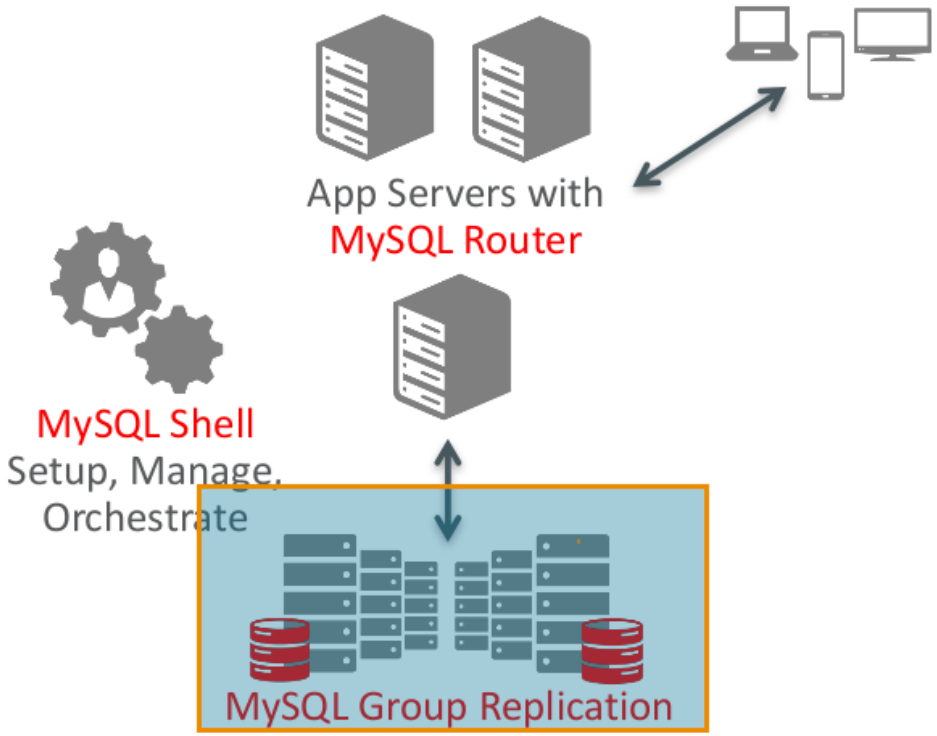
- @lefred
- MySQL Evangelist
- Managing MySQL since 3.23
- devops believer
- living in Belgium **B E**
- https://lefred.be

ORACLE

Group Replication: heart of MySQL InnoDB Cluster



Group Replication: heart of MySQL InnoDB Cluster



MySQL Group Replication

but what is it ?!?

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- Paxos based protocol (our implementation is close to Mencius)

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- Paxos based protocol (our implementation is close to Mencius)
- GR allows to write on all Group Members (cluster nodes) simultaneously while retaining consistency

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- Paxos based protocol (our implementation is close to Mencius)
- GR allows to write on all Group Members (cluster nodes) simultaneously while retaining consistency
- GR implements conflict detection and resolution

MySQL Group Replication

but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- Paxos based protocol (our implementation is close to Mencius)
- GR allows to write on all Group Members (cluster nodes) simultaneously while retaining consistency
- GR implements conflict detection and resolution
- GR allows automatic distributed recovery

MySQL Group Replication

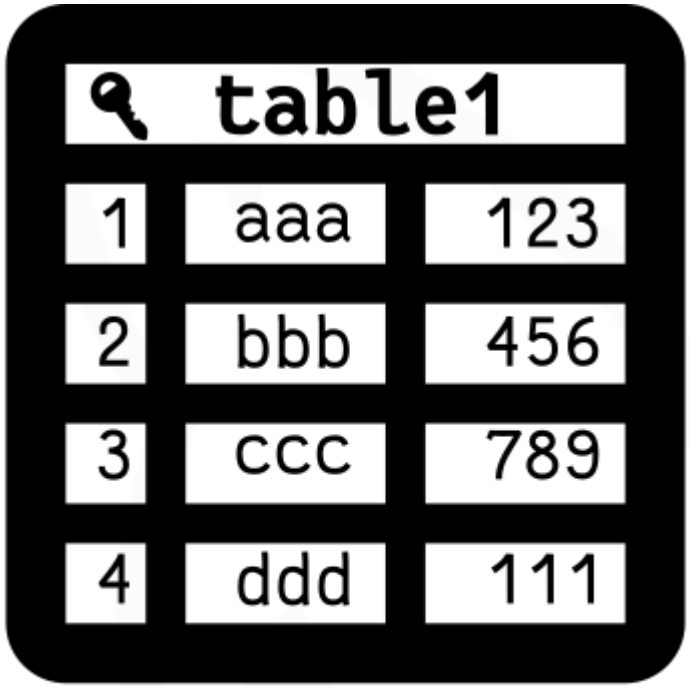
but what is it ?!?

- GR is a plugin for MySQL, made by MySQL and packaged with MySQL
- GR is an implementation of Replicated Database State Machine theory
- Paxos based protocol (our implementation is close to Mencius)
- GR allows to write on all Group Members (cluster nodes) simultaneously while retaining consistency
- GR implements conflict detection and resolution
- GR allows automatic distributed recovery
- Supported on **all MySQL platforms** !!
  - Linux, Windows, Solaris, OSX, FreeBSD

terminology

Write vs Writeset

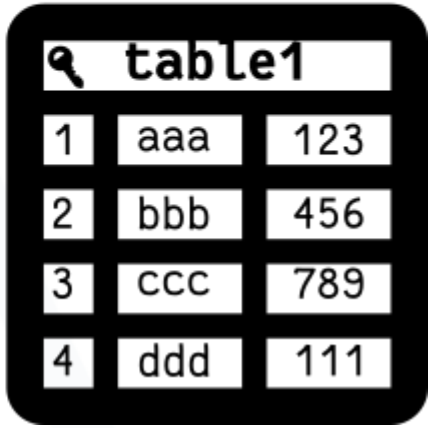
Let's illustrate a table:




```
CREATE TABLE `table1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `b` char(20) NOT NULL DEFAULT '',
  `c` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB
```

Now let's make a change

```
start transaction;
update table1 set c = 999 where id =2;
update table1 set b = "eee" where id = 3;
commit;
```



and at commit time:

```
start transaction;
update table1 set c = 999 where id =2;
update table1 set b = "eee" where id = 3;
commit;
```

Writesets

Contain the hash for the rows PKs that are changed and in some cases the hashes of foreign keys or others dependencies that need to be captured (e.g. non NULL UKs). Writesets are gathered during transaction execution.

Writesets

Contain the hash for the rows PKs that are changed and in some cases the hashes of foreign keys or others dependencies that need to be captured (e.g. non NULL UKs). Writesets are gathered during transaction execution.

Writes

Called also write values, refers to the actual changes. Write values are also gathered during transaction execution.

Writeset - examples

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+
```

Writeset - examples

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t2 values (1,2);
```

Writeset - examples

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t2 values (1,2);

pke: PRIMARY | test | t2 | 1 | 1     hash: 11853456929268668462
```

Writeset - examples

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t2 values (1,2);

pke: PRIMARY | test | t2 | 1 | 1     hash: 11853456929268668462

mysql> update t2 set name=3 where id=1;
```

Writeset - examples

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t2 values (1,2);

pke: PRIMARY | test | t2 | 1 | 1     hash: 11853456929268668462

mysql> update t2 set name=3 where id=1;

pke: PRIMARY | test | t2 | 1 | 1    hash: 10002085147685770725
pke: PRIMARY | test | t2 | 1 | 1    hash: 10002085147685770725
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t3 values (1,2,3);
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t3 values (1,2,3);

pke: PRIMARY | test |t3 | 1 | 1    hash: 79134815725924853
pke: name    | test |t3 | 2       hash: 11034644986657565827
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t3 values (1,2,3);

pke: PRIMARY | test |t3 | 1 | 1    hash: 79134815725924853
pke: name    | test |t3 | 2       hash: 11034644986657565827

mysql> update t3 set name=3 where id=1;
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t3 values (1,2,3);

pke: PRIMARY | test |t3 | 1 | 1    hash: 79134815725924853
pke: name    | test |t3 | 2       hash: 11034644986657565827

mysql> update t3 set name=3 where id=1;

pke: PRIMARY | test | t3 | 1 | 1    hash: 79134815725924853
pke: name    | test | t3 | 3       hash: 18082071075512932388
pke: PRIMARY | test | t3 | 1 | 1    hash: 79134815725924853
pke: name    | test | t3 | 2       hash: 11034644986657565827
```

Writeset - examples (2)

```
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| id    | binary(1) | NO   | PRI | NULL    |       |
| name  | binary(2) | YES  | UNI | NULL    |       |
| name2 | binary(1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+

mysql> insert into t3 values (1,2,3);

pke: PRIMARY | test |t3 | 1 | 1    hash: 79134815725924853
pke: name    | test |t3 | 2       hash: 11034644986657565827

mysql> update t3 set name=3 where id=1;

pke: PRIMARY | test | t3 | 1 | 1    hash: 79134815725924853
pke: name    | test | t3 | 3       hash: 18082071075512932388
pke: PRIMARY | test | t3 | 1 | 1    hash: 79134815725924853
pke: name    | test | t3 | 2       hash: 11034644986657565827

[after image]
[before image]
```

GR is nice, but how does it work ?

GR is nice, but how does it work ?

it´s just ...

GR is nice, but how does it work ?

it´s just ...

GR is nice, but how does it work ?

it´s just ...

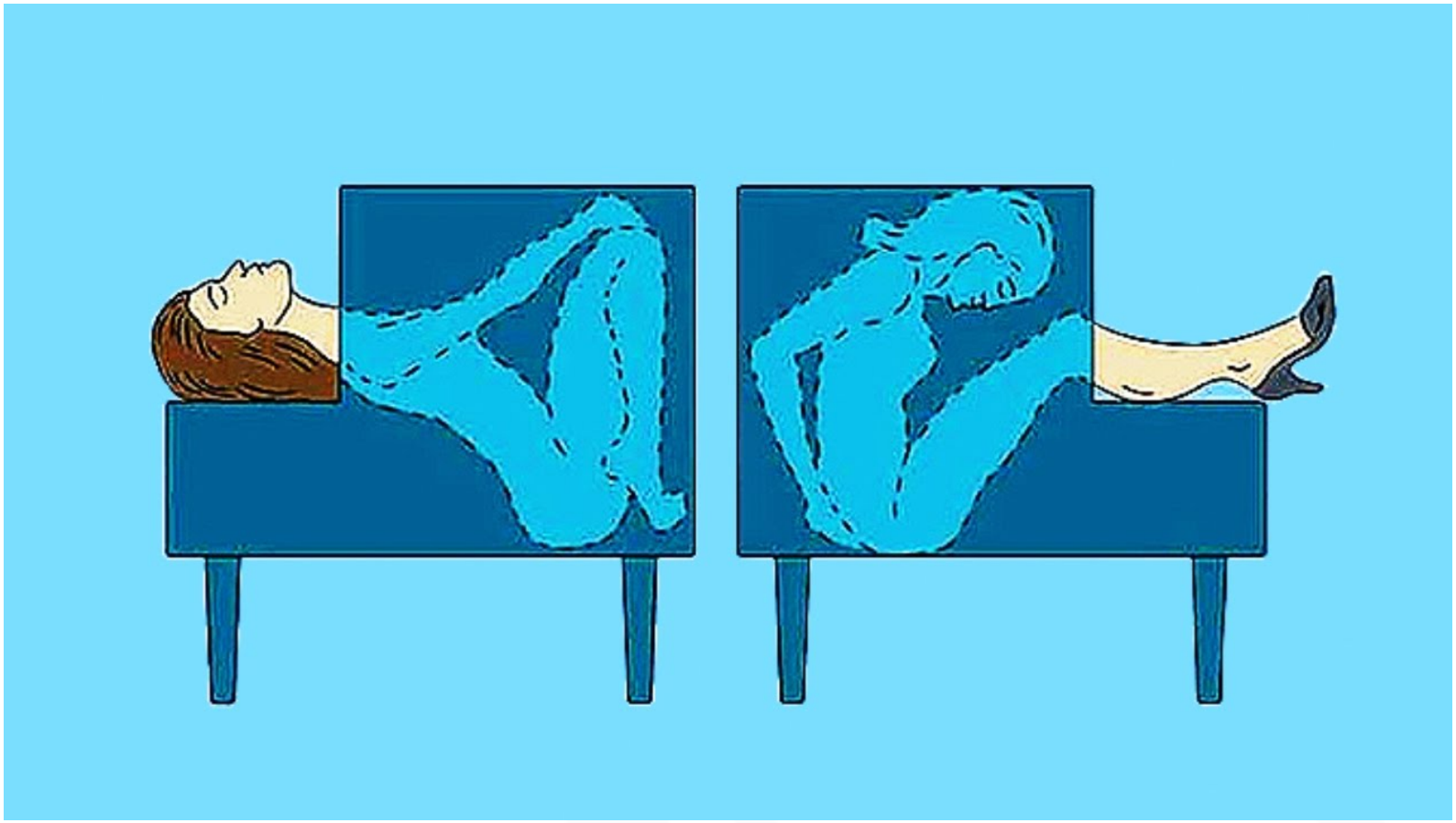
... no, in fact the writesets replication is **synchronous** and then certification and apply of the changes are local to each nodes and asynchronous.

GR is nice, but how does it work ?

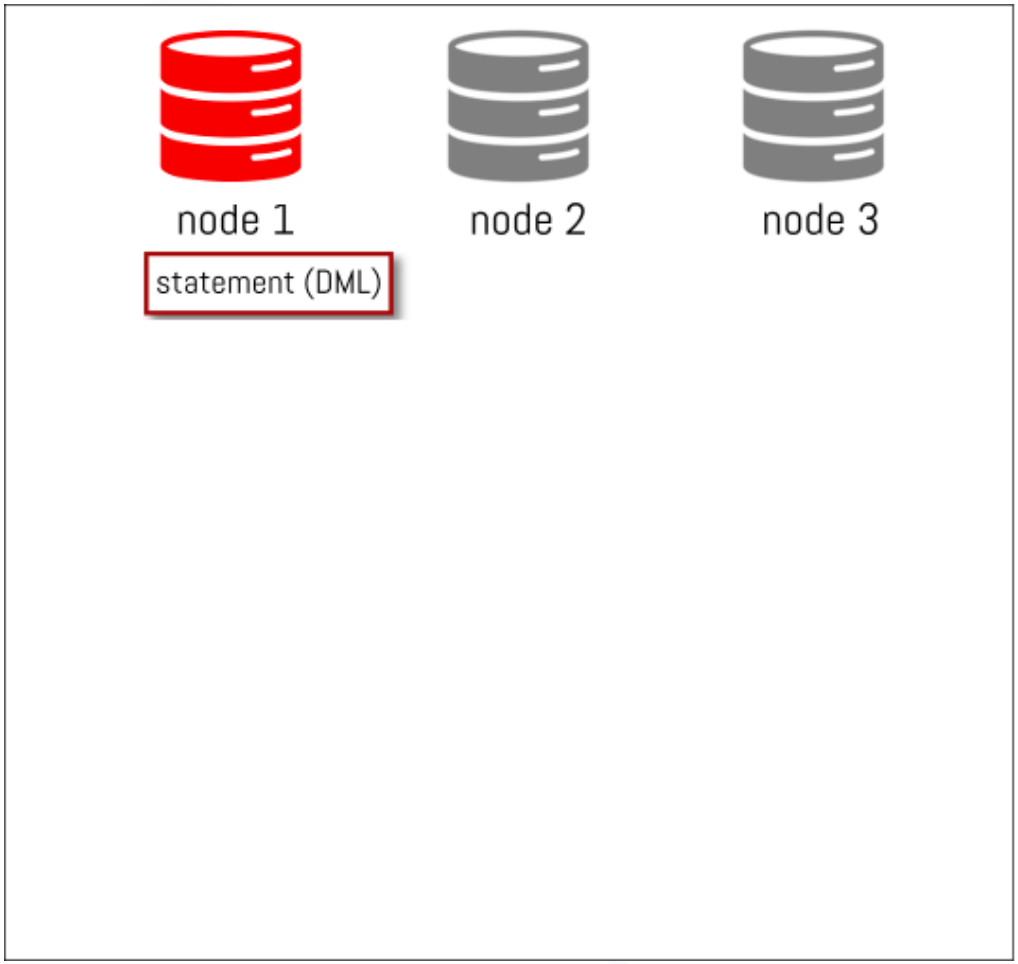
it´s just ...

... no, in fact the writesets replication is **synchronous** and then certification and apply of the changes are local to each nodes and asynchronous.

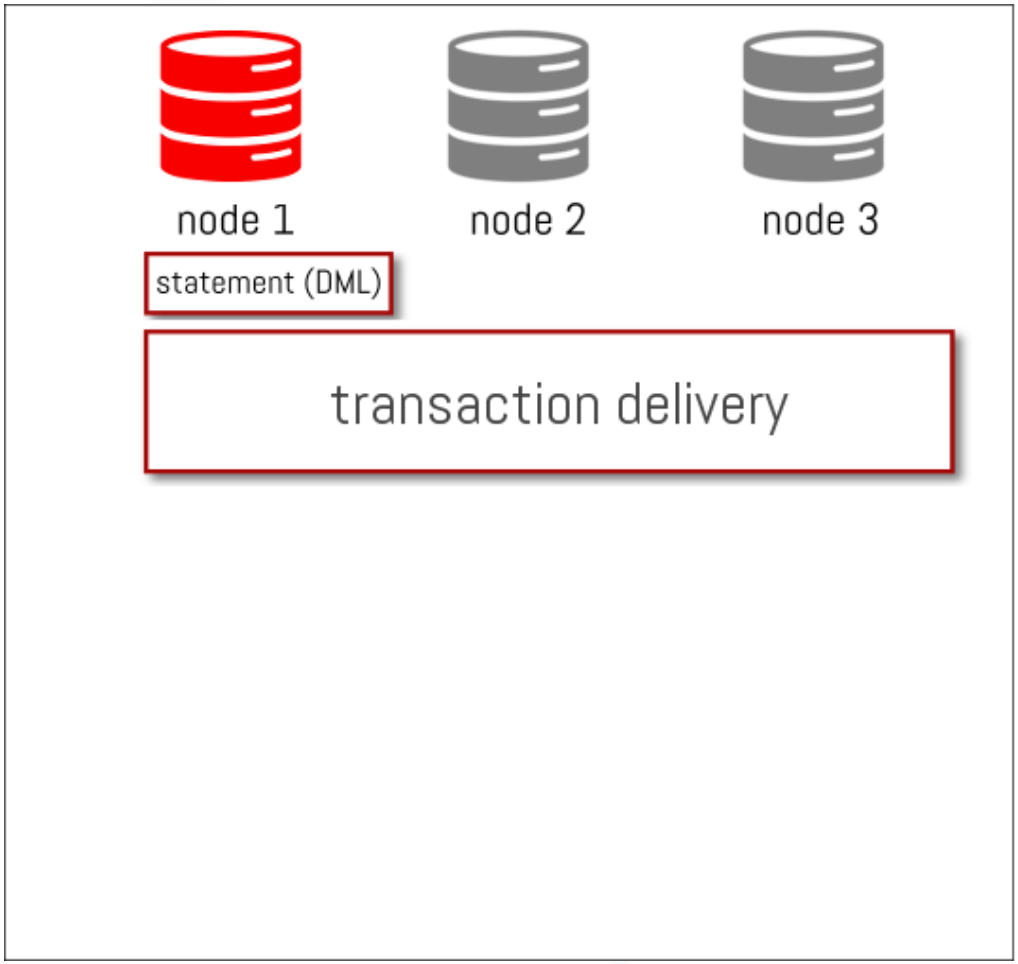
not that easy to understand... right ? As a picture is worth a 1000 words, let´s illustrate this...



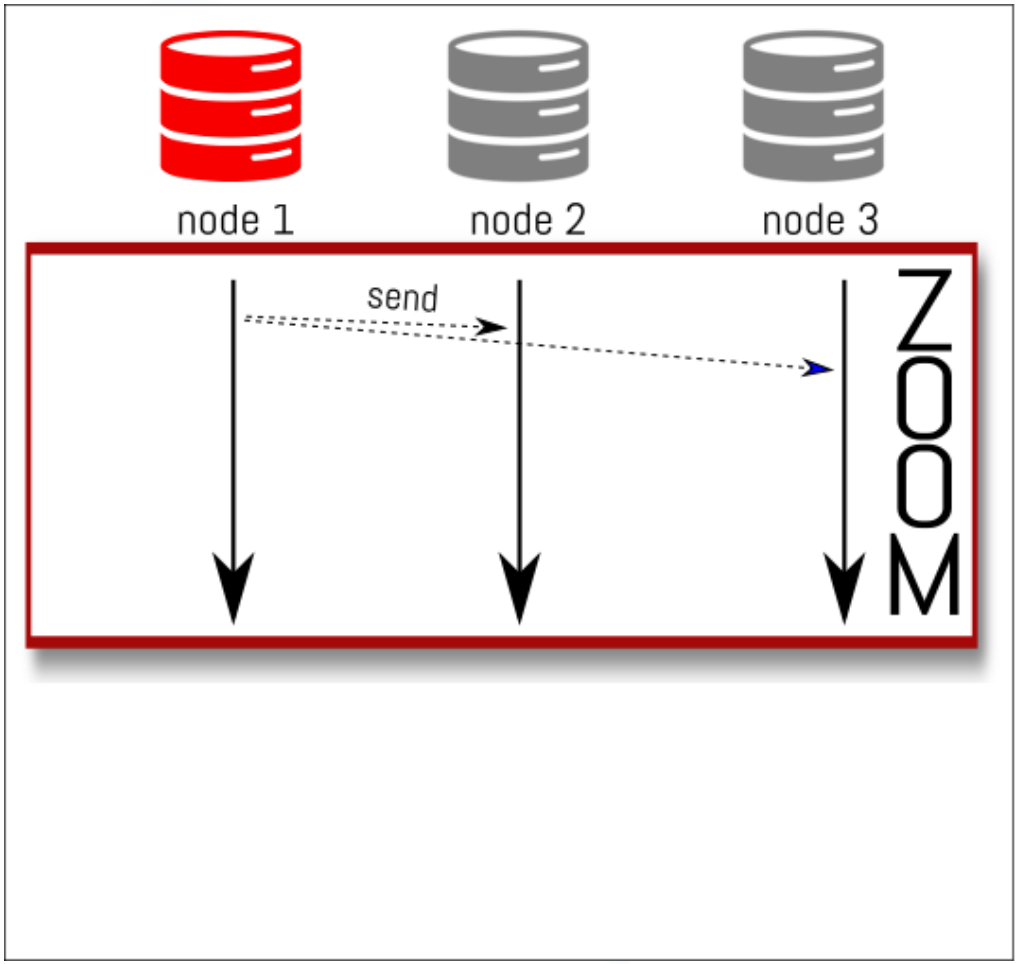
MySQL Group Replication



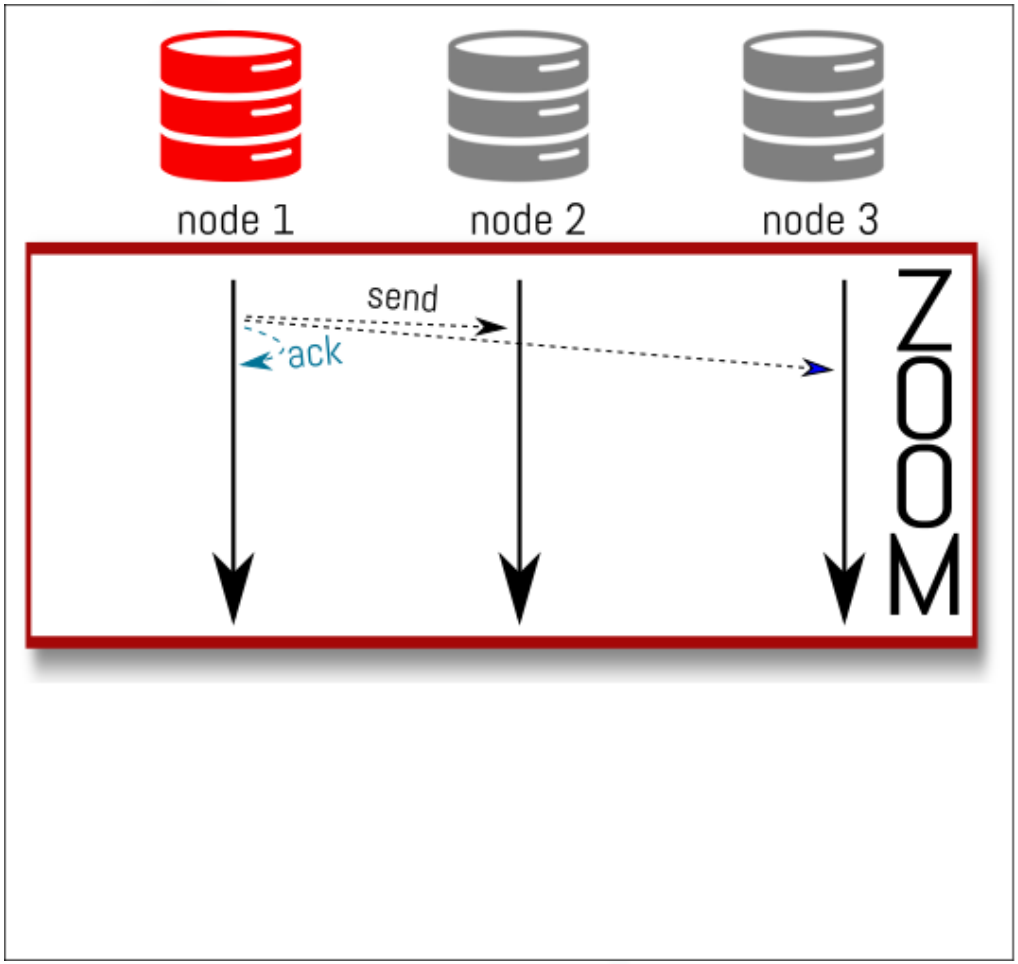
MySQL Group Replication



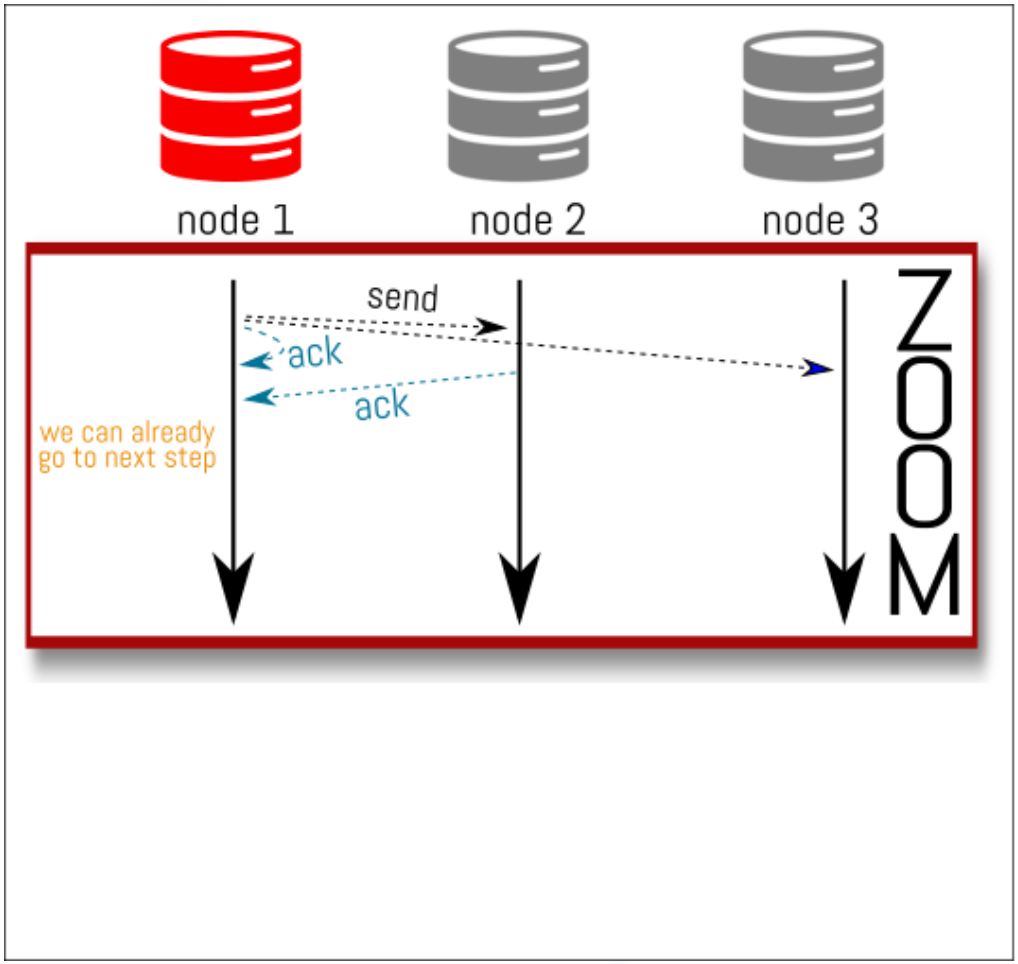
MySQL Group Replication



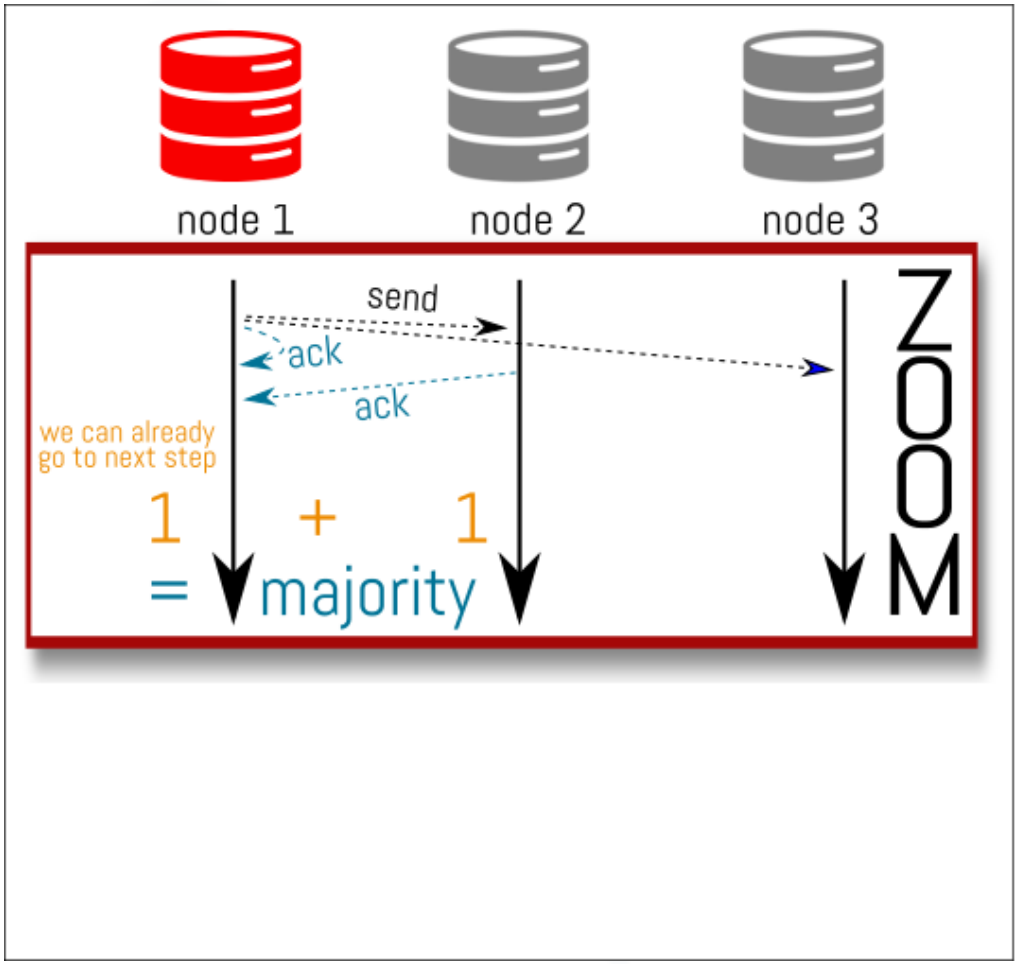
MySQL Group Replication



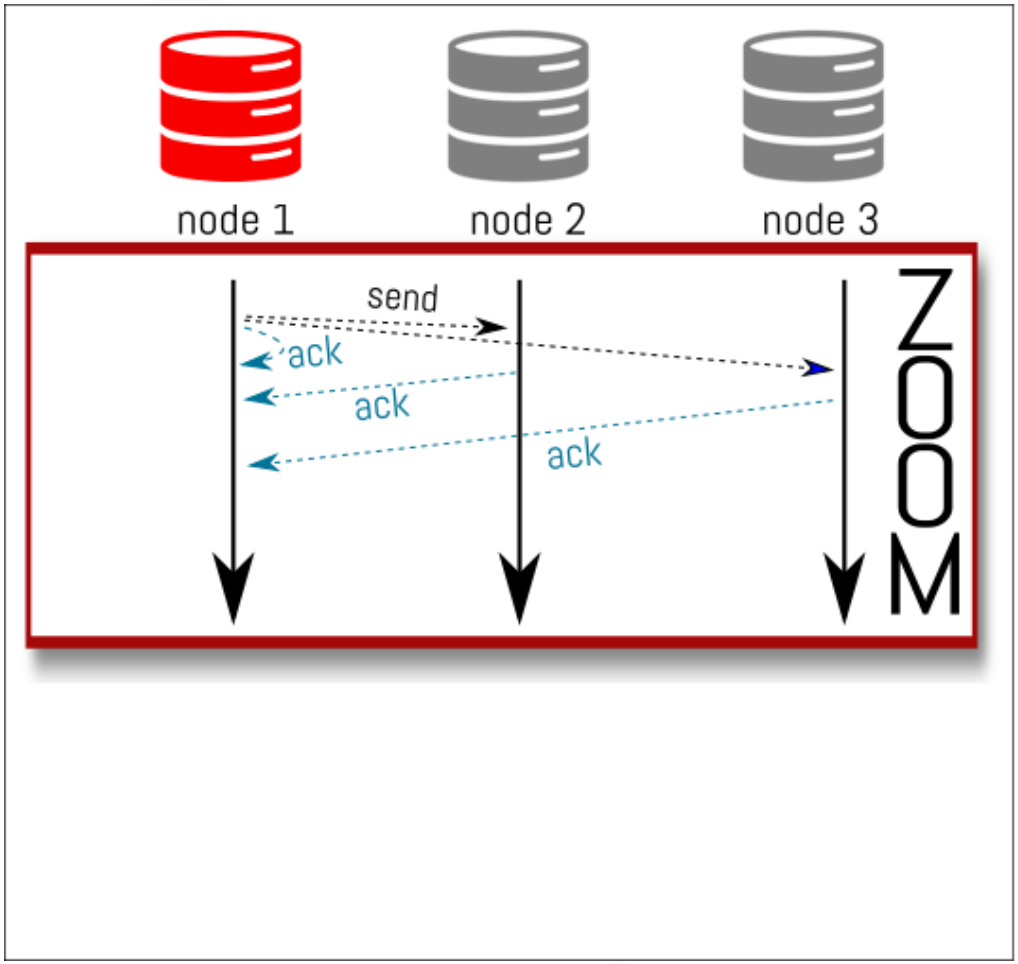
MySQL Group Replication



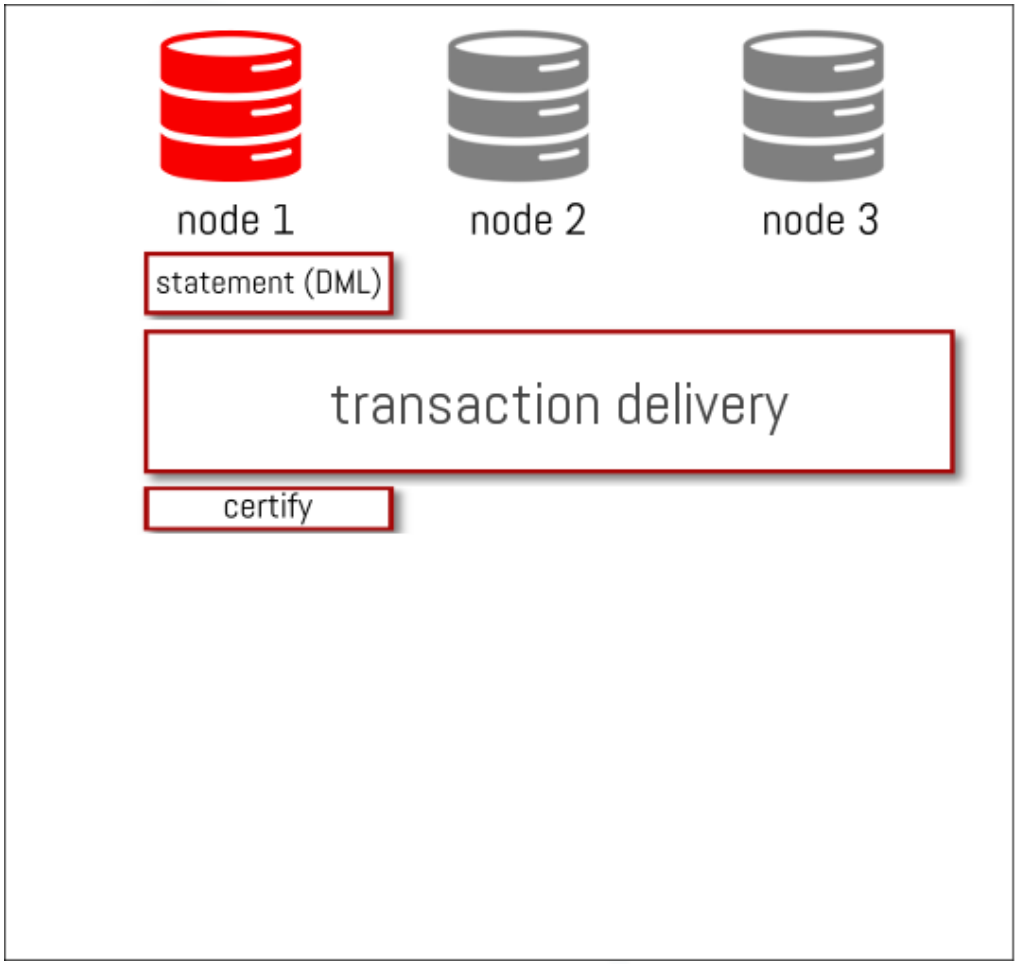
MySQL Group Replication



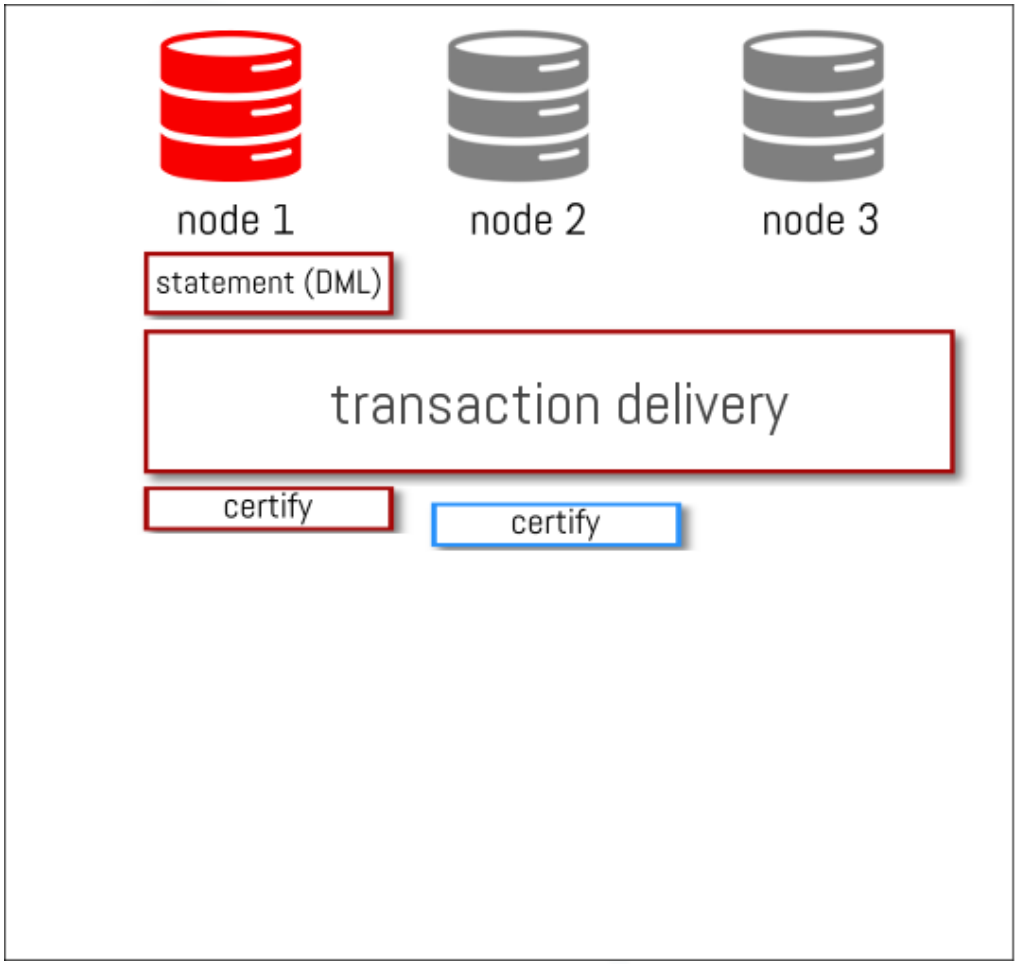
MySQL Group Replication



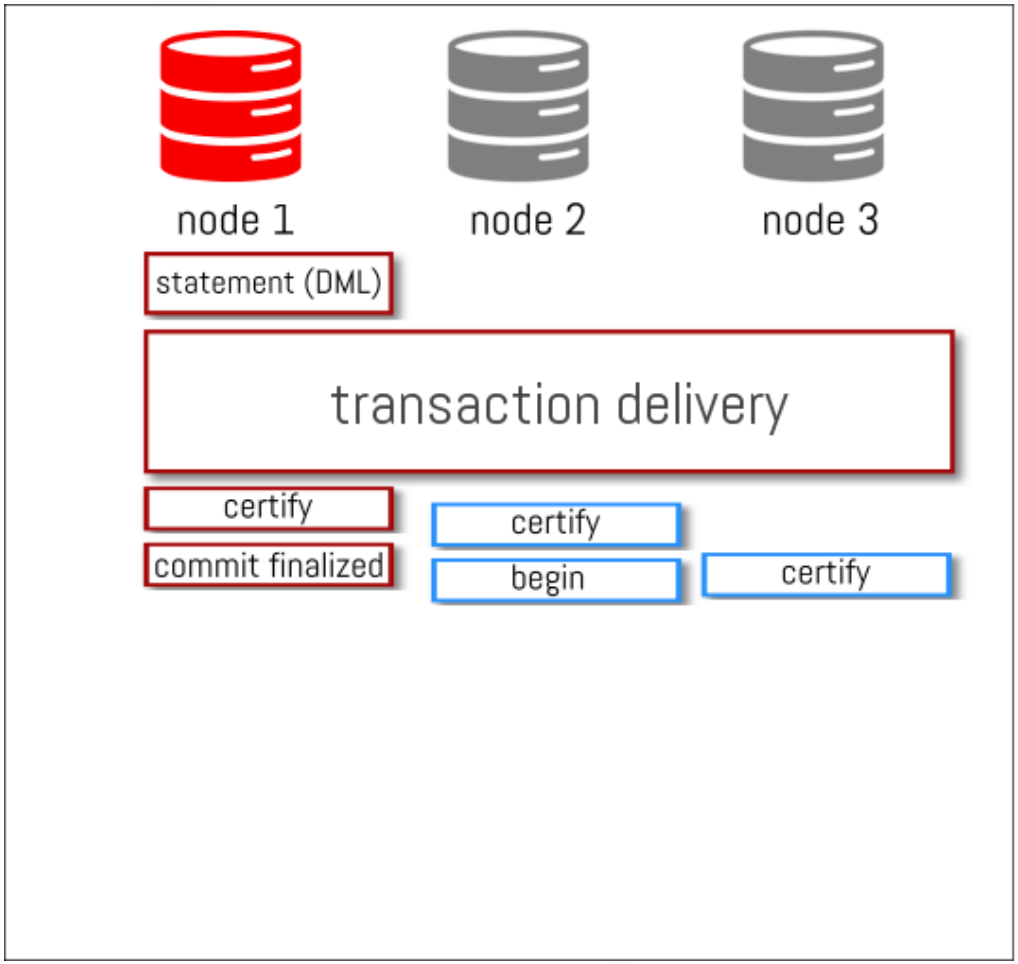
MySQL Group Replication



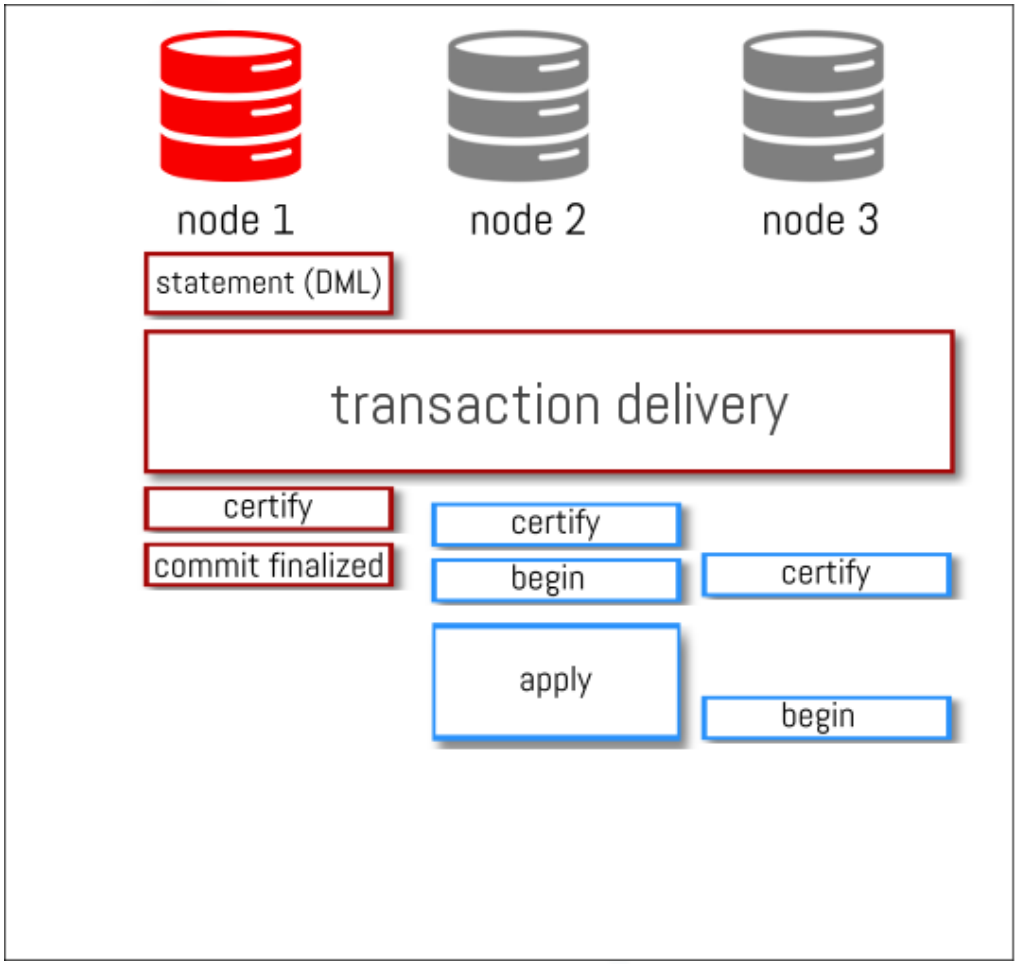
MySQL Group Replication



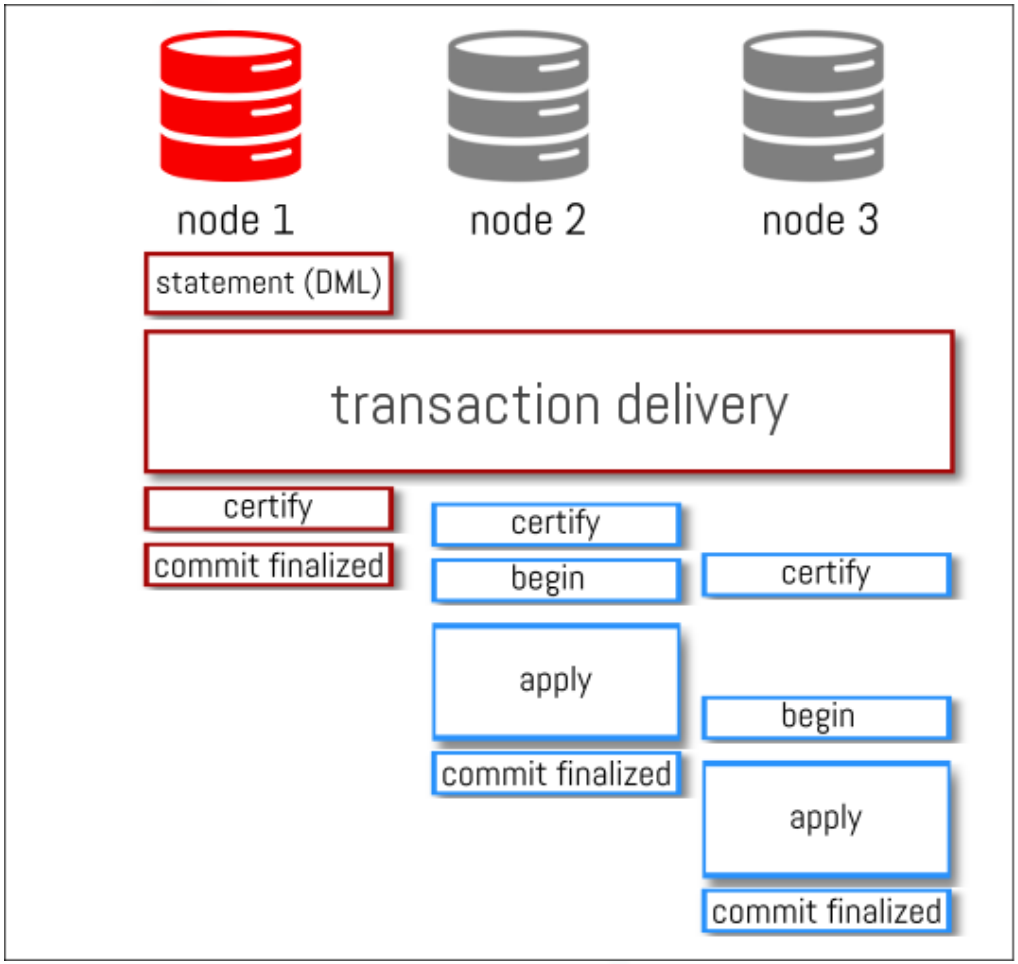
MySQL Group Replication



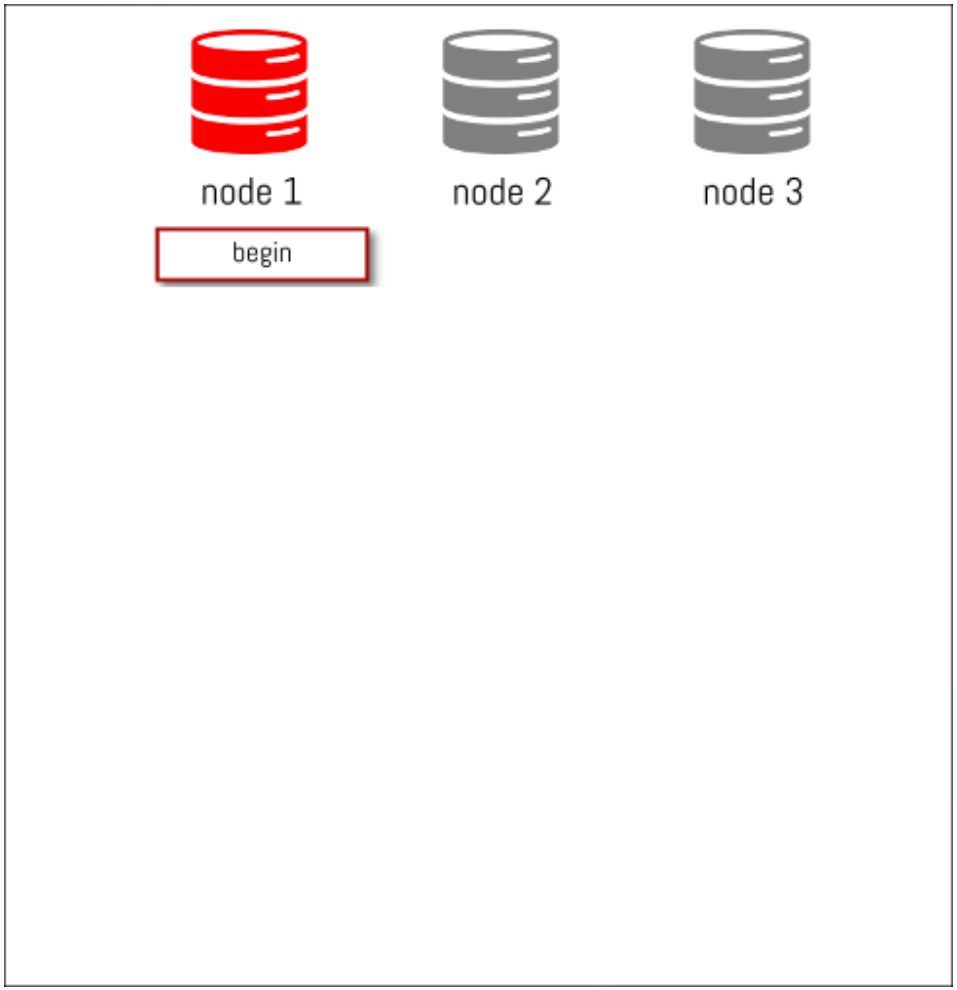
MySQL Group Replication



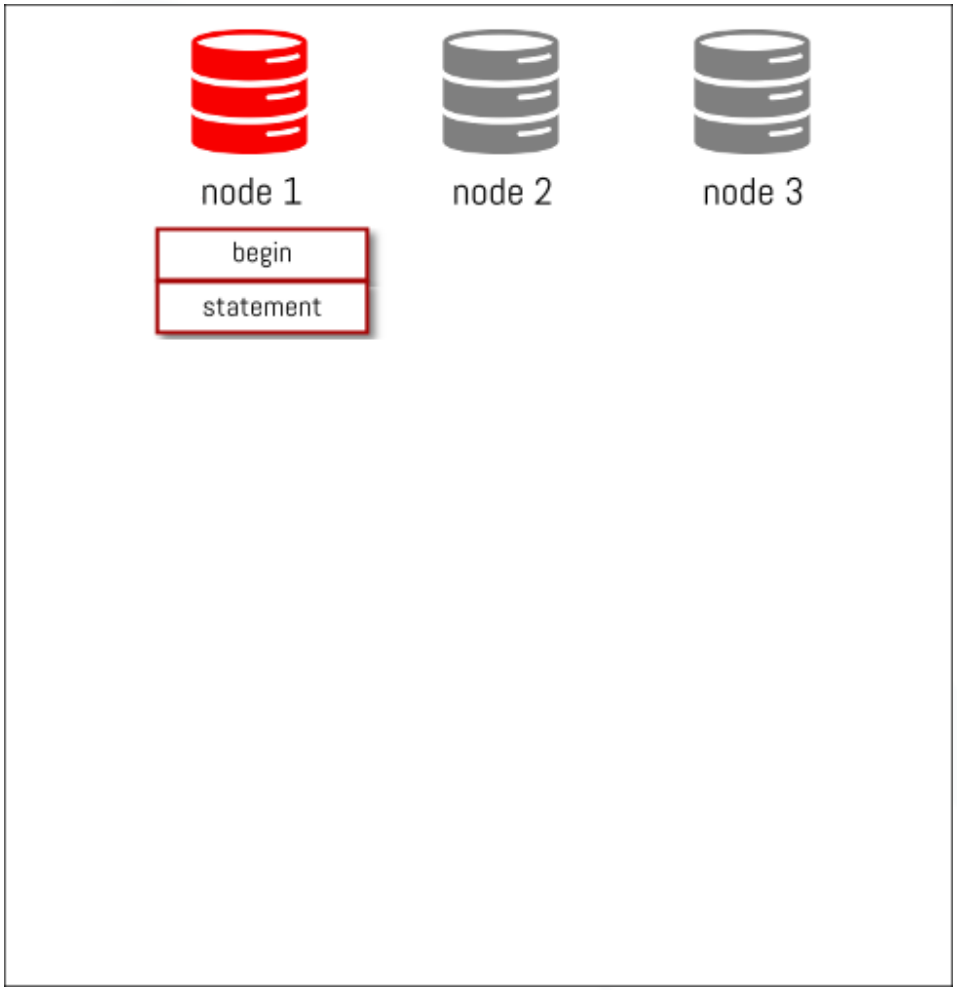
MySQL Group Replication



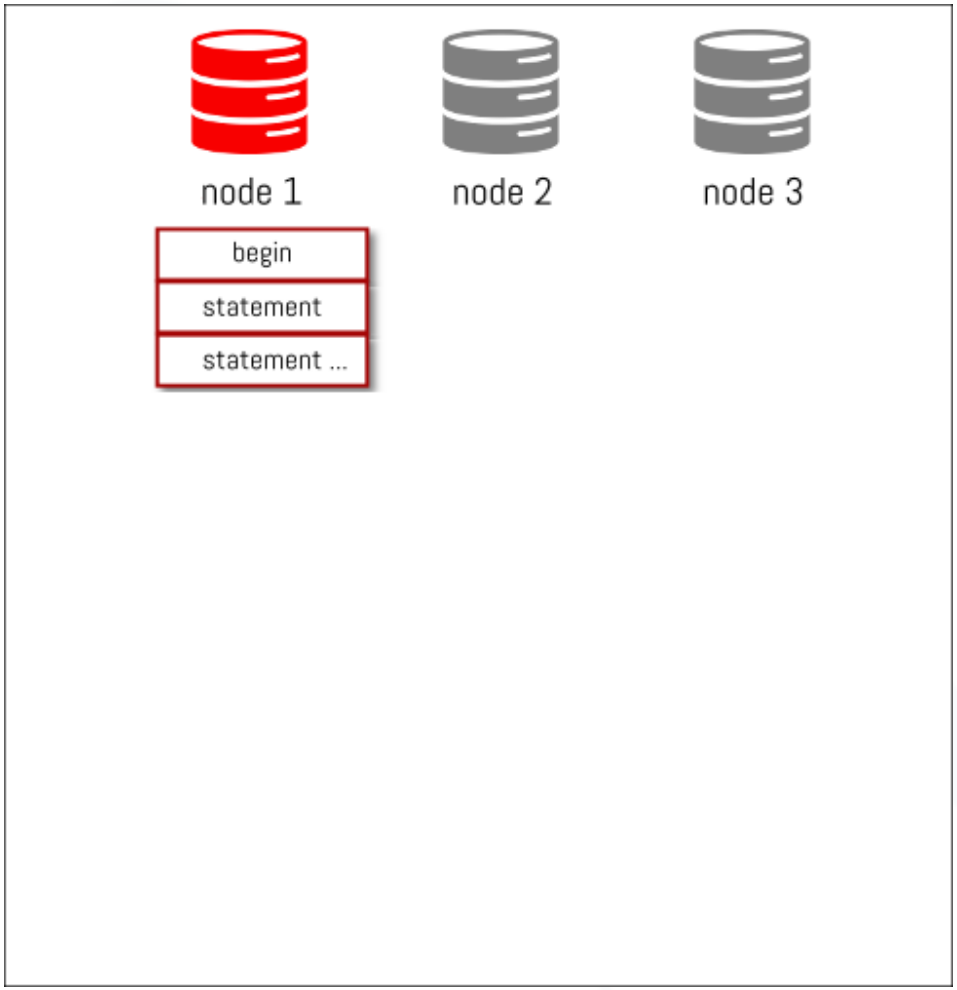
MySQL Group Replication (full transaction)



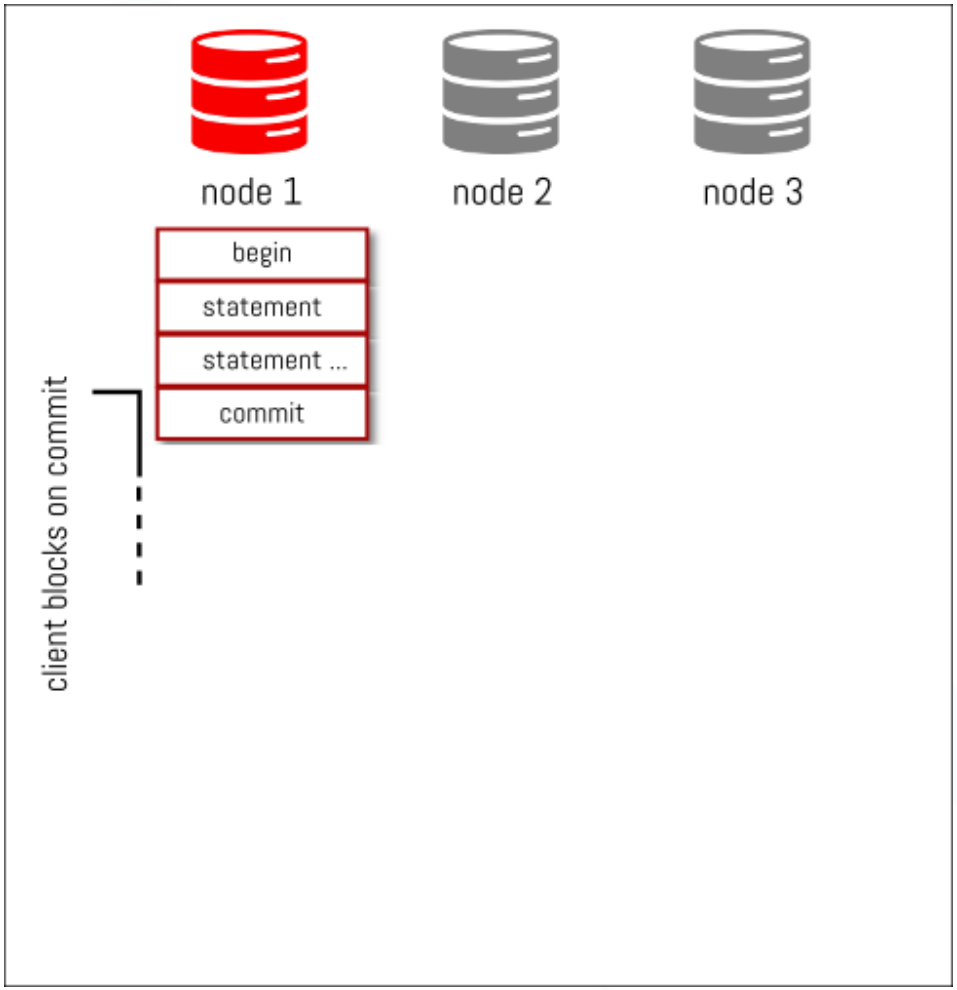
MySQL Group Replication (full transaction)



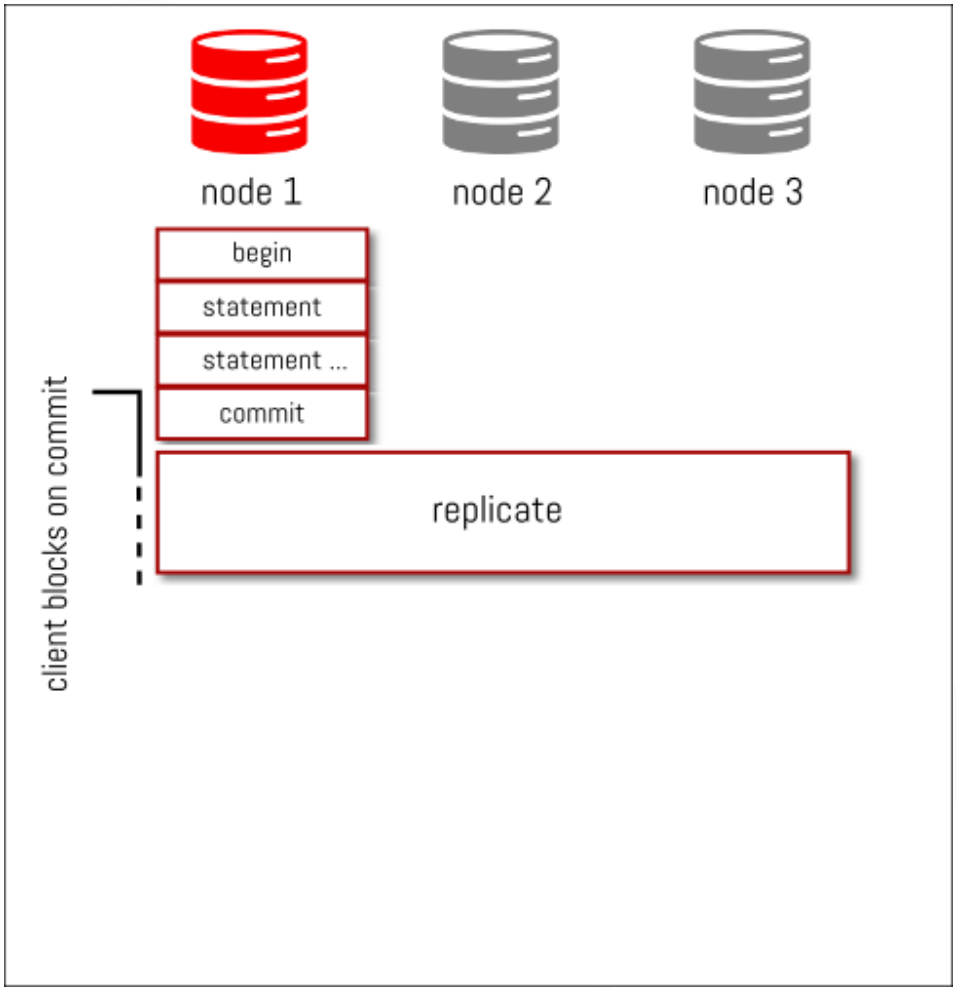
MySQL Group Replication (full transaction)



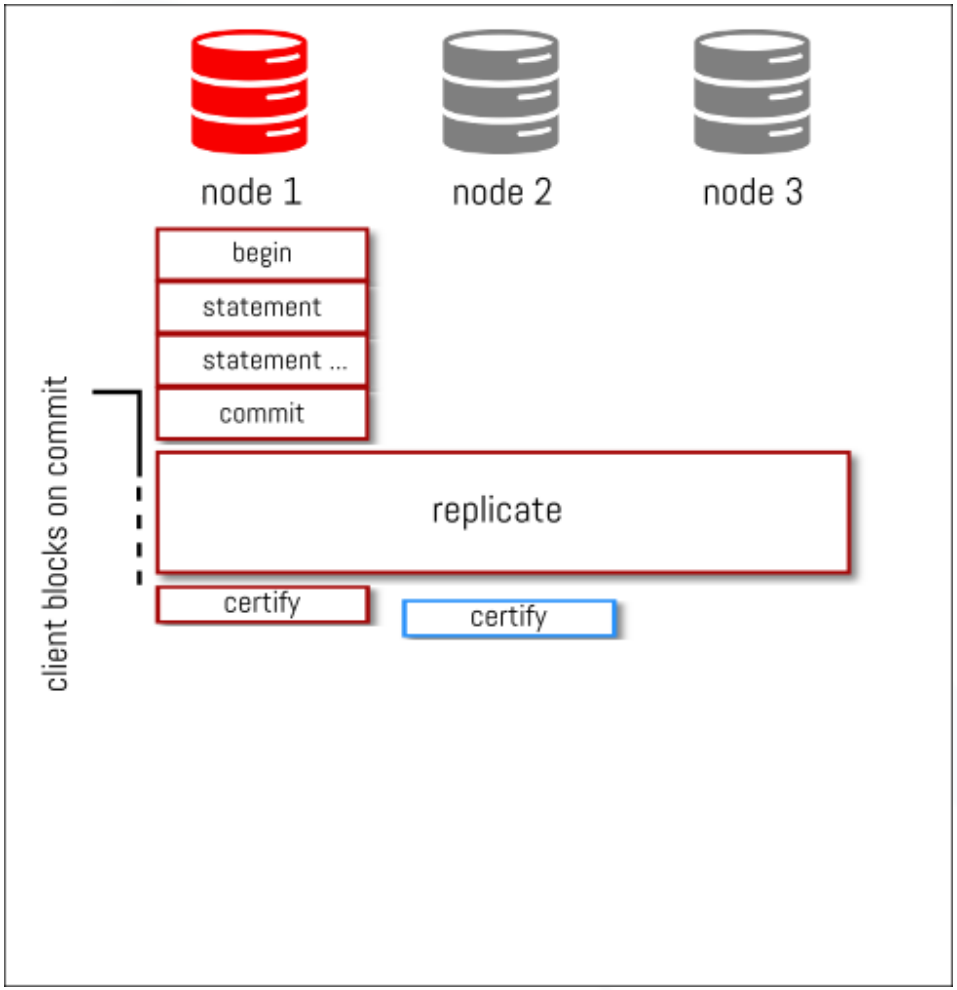
MySQL Group Replication (full transaction)



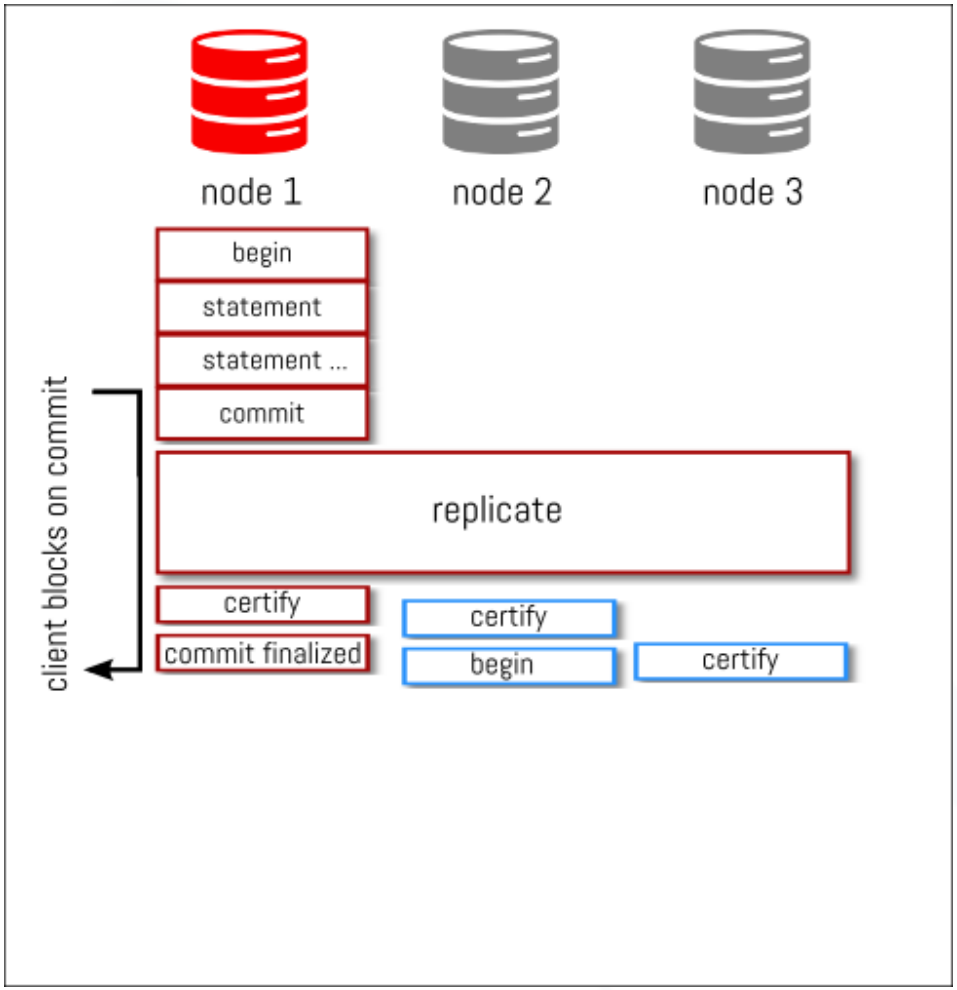
MySQL Group Replication (full transaction)



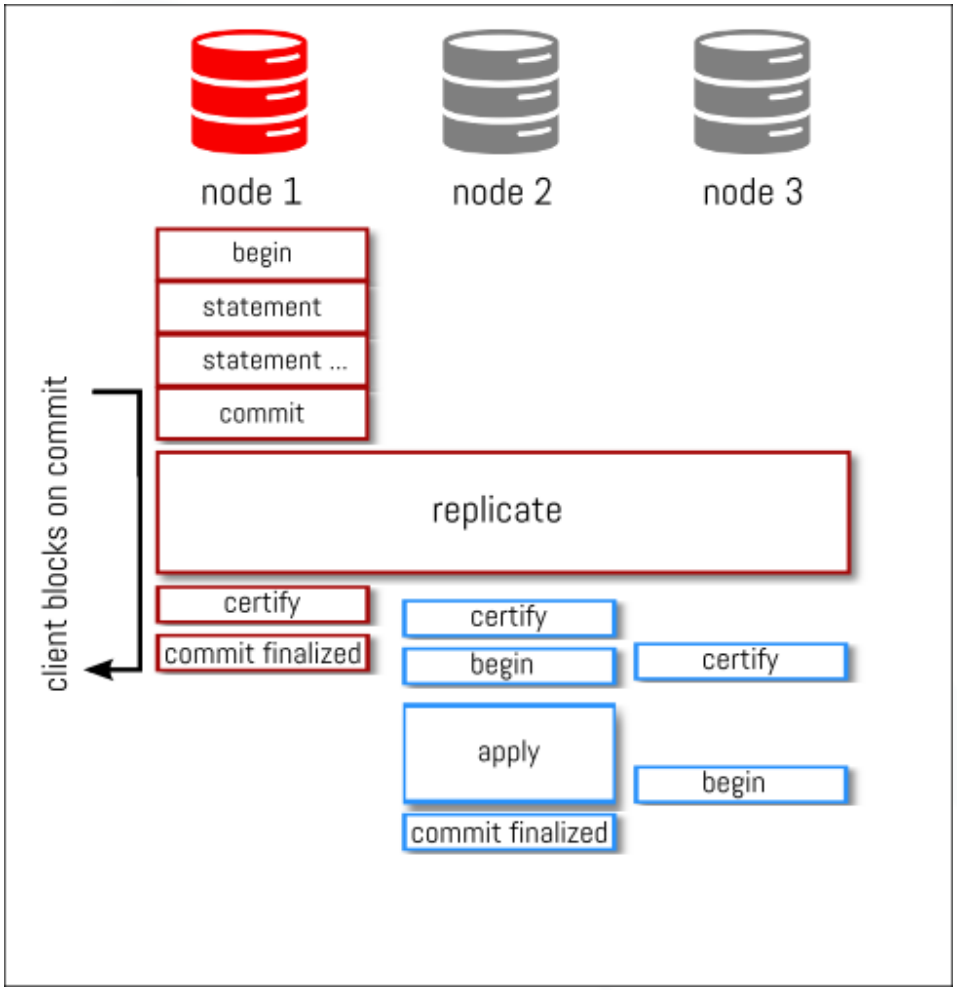
MySQL Group Replication (full transaction)



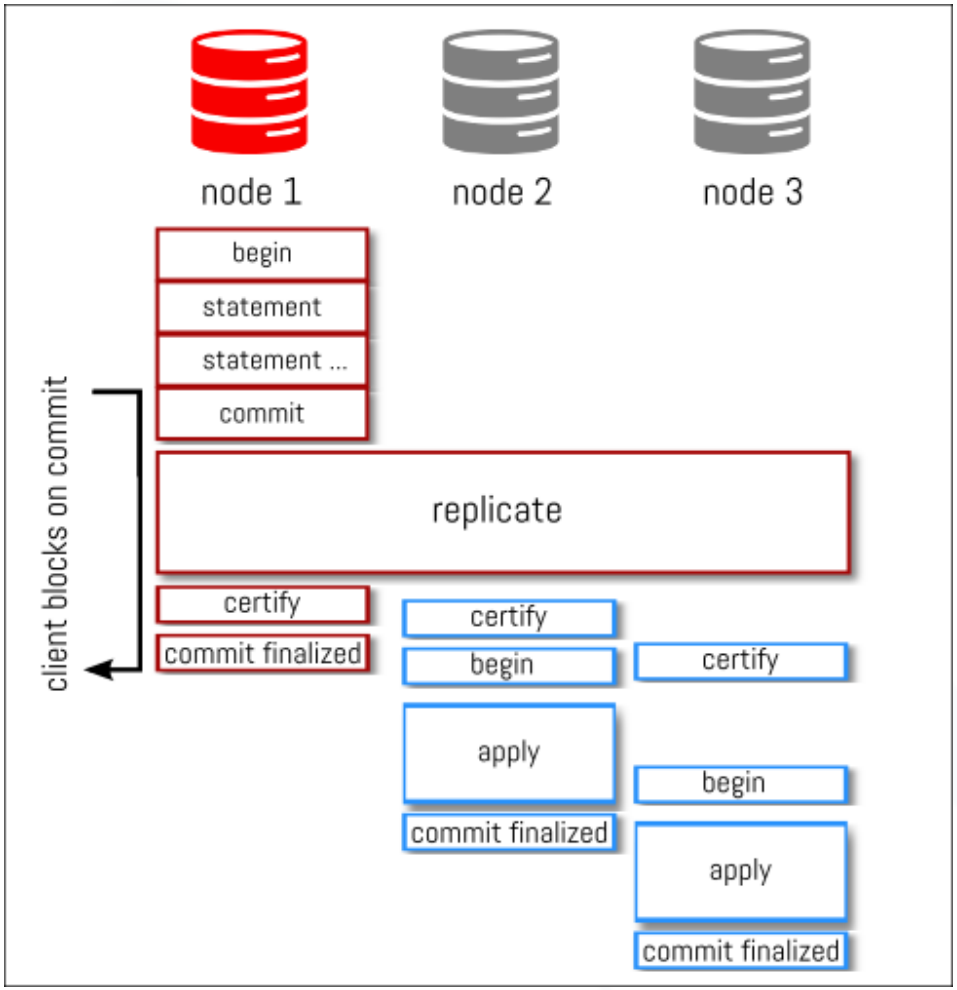
MySQL Group Replication (full transaction)



MySQL Group Replication (full transaction)



MySQL Group Replication (full transaction)



MySQL Group Communication System (GCS)

- MySQL Xcom protocol

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol
- its task: *deliver messages across the distributed system*.

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol
- its task: *deliver messages across the distributed system*:
  - atomically

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol
- its task: *deliver messages across the distributed system*:
  - atomically
  - in Total Order

MySQL Group Communication System (GCS)

- MySQL Xcom protocol
- Replicated Database State Machine
- Paxos based protocol
- its task: *deliver messages across the distributed system*:
  - atomically
  - in Total Order
- MySQL Group Replication receives the Ordered 'tickets' from this GCS subsystem.

Total Order

GTID generation

How does Group Replication handle GTIDs ?

There are two ways of generating GTIDs:

How does Group Replication handle GTIDs ?

There are two ways of generating GTIDs:

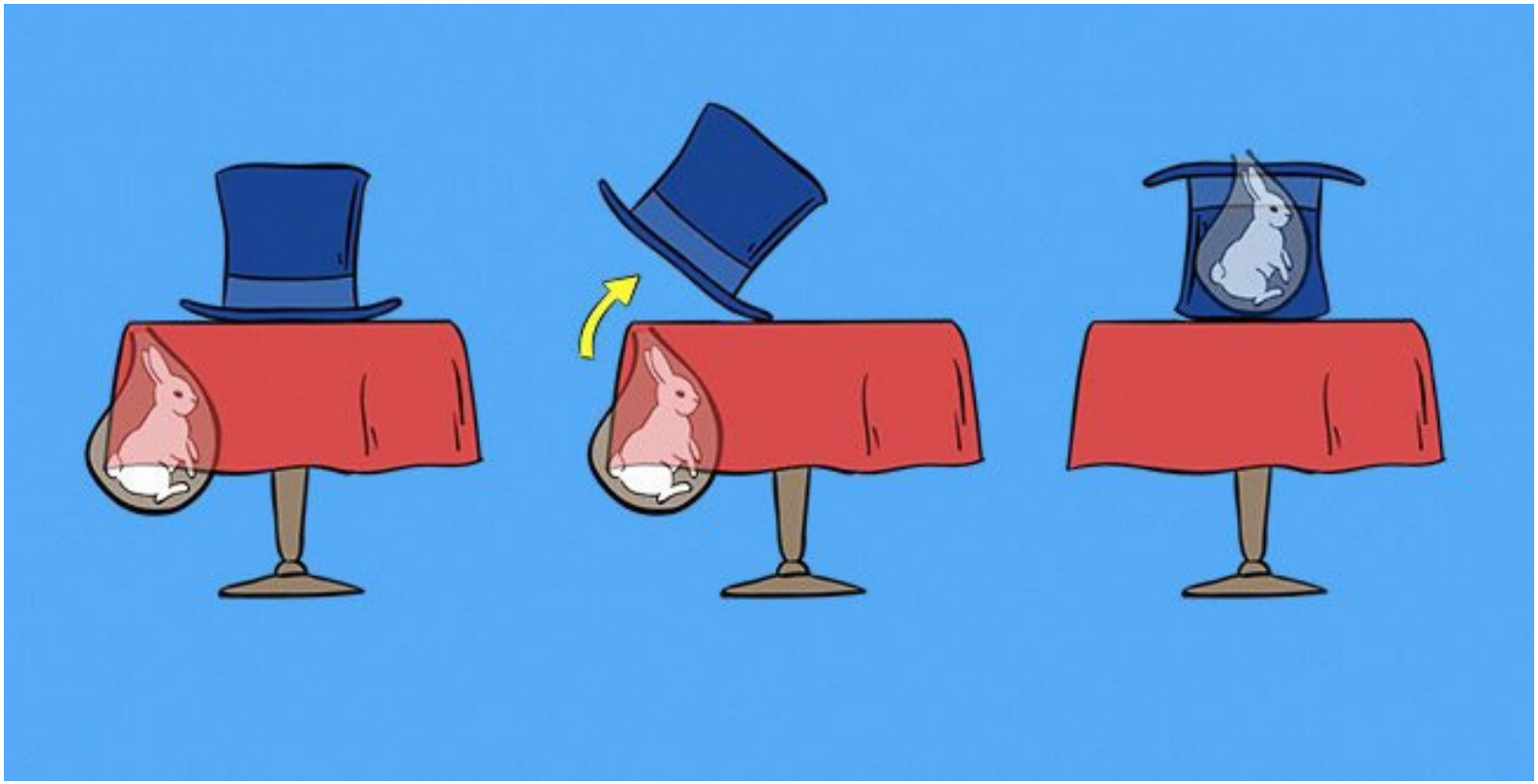
- **AUTOMATIC**: the transaction is assigned with an automatically generated id during commit. Where regular replication uses the source server UUID, on Group Replication, the group name is used.

How does Group Replication handle GTIDs ?

There are two ways of generating GTIDs:

- **AUTOMATIC**: the transaction is assigned with an automatically generated id during commit. Where regular replication uses the source server UUID, on Group Replication, the group name is used.

- **ASSIGNED**: the user assigns manually a GTID through `SET GTID_NEXT` to the transaction. This is common to any replication format and the id is assigned before the transaction starts.

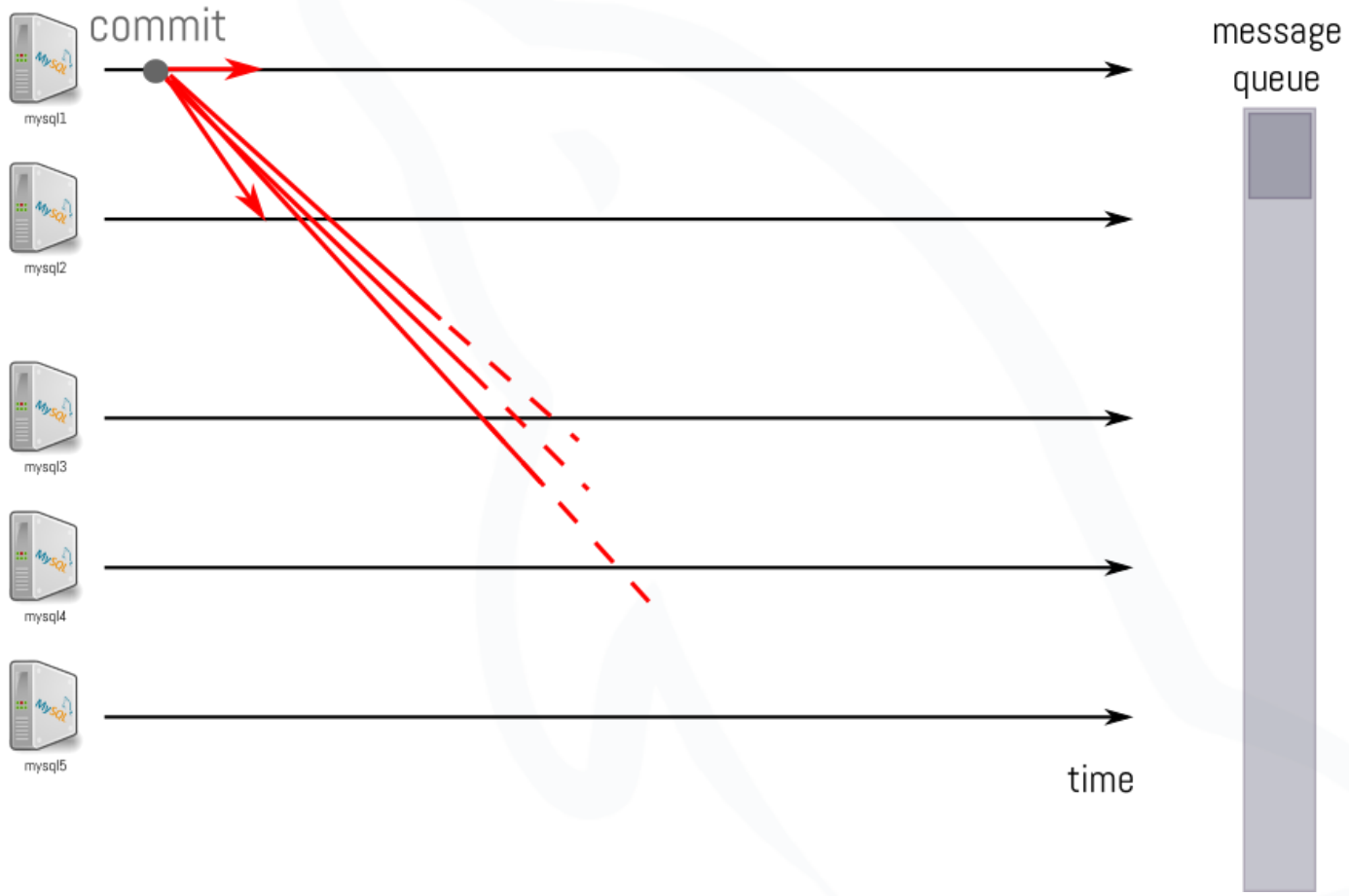


Group Replication : Total Order Delivery - GTID

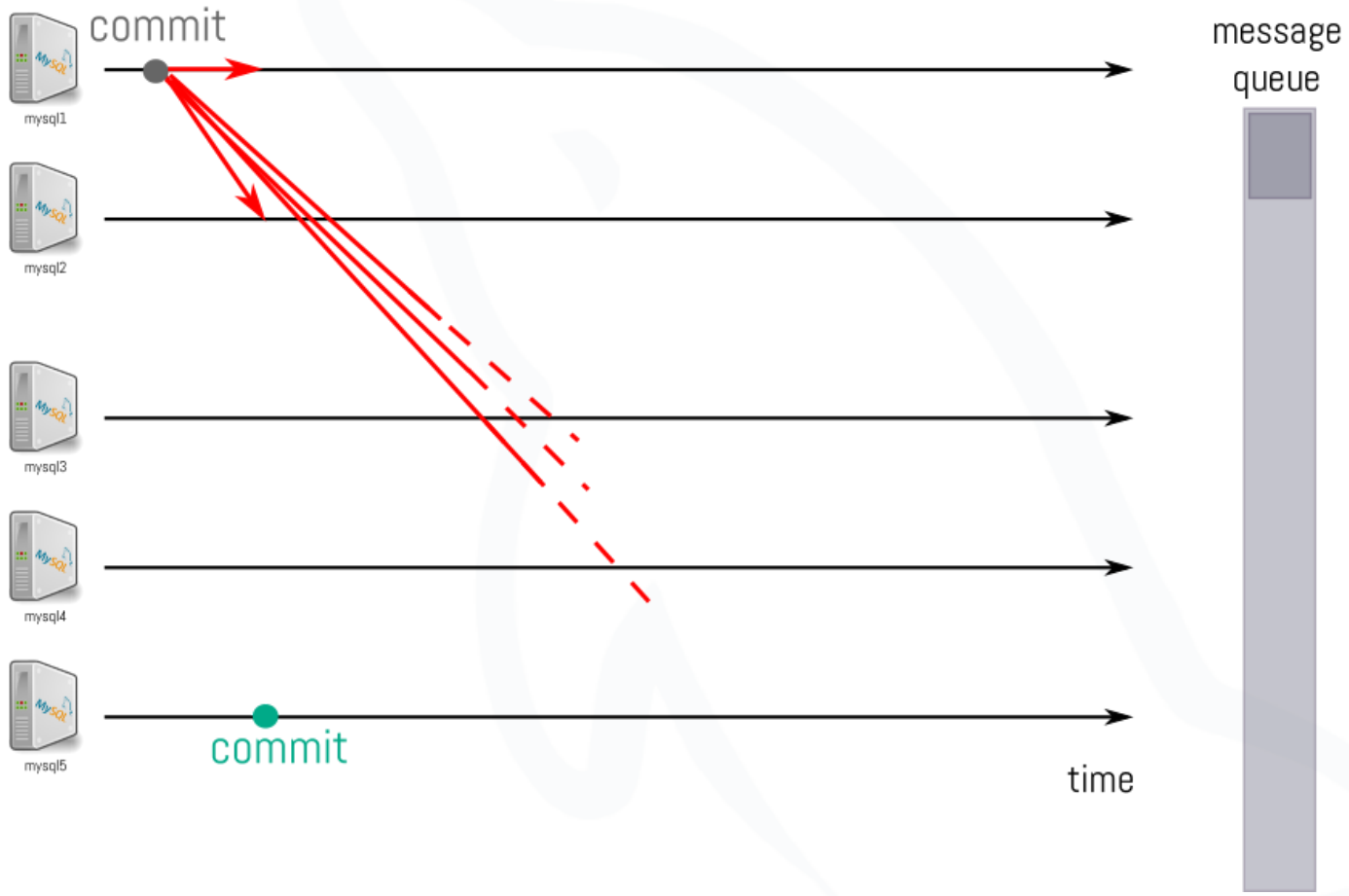
Group Replication : Total Order Delivery - GTID

Group Replication : Total Order Delivery - GTID

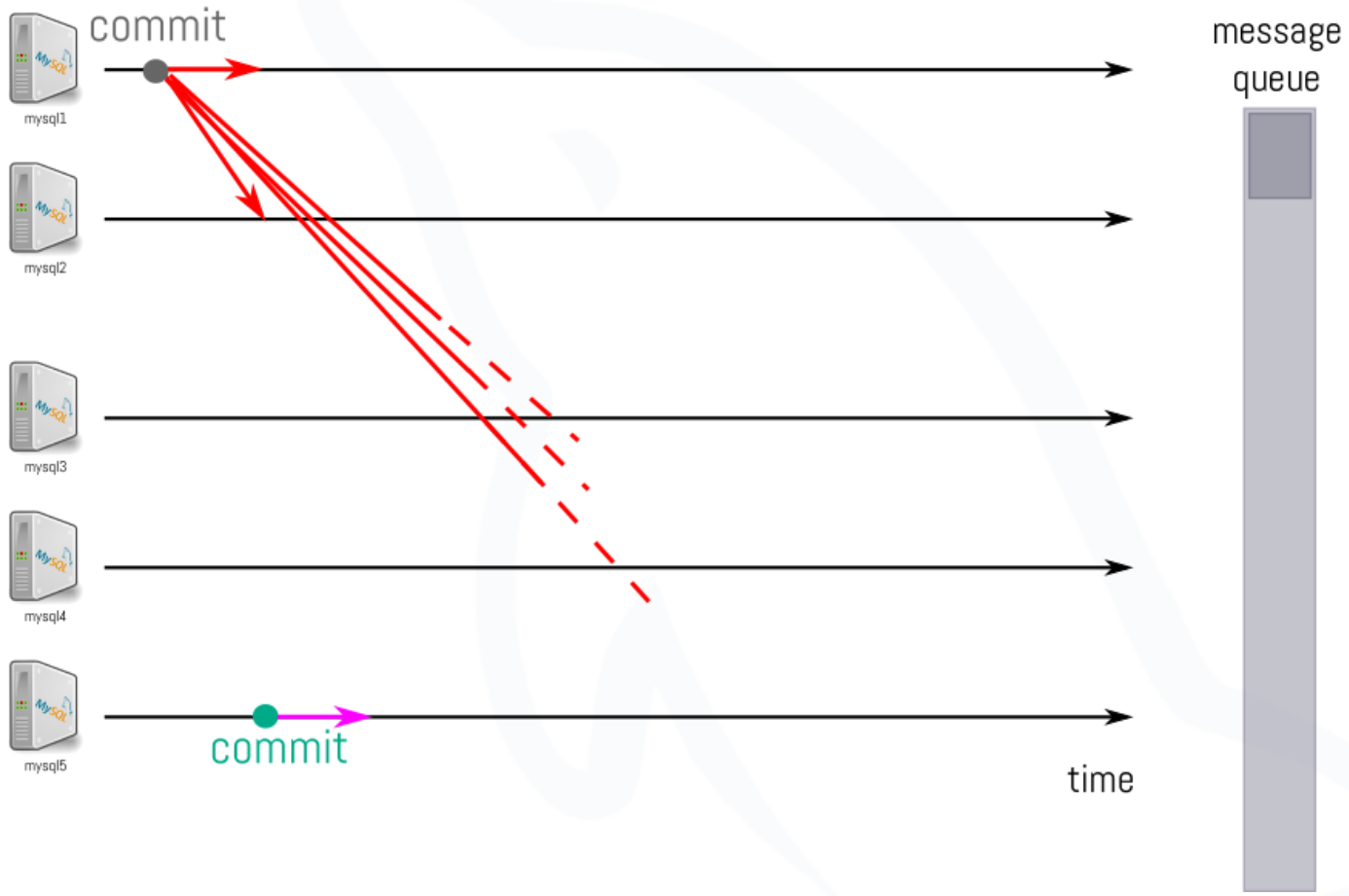
Group Replication : Total Order Delivery - GTID



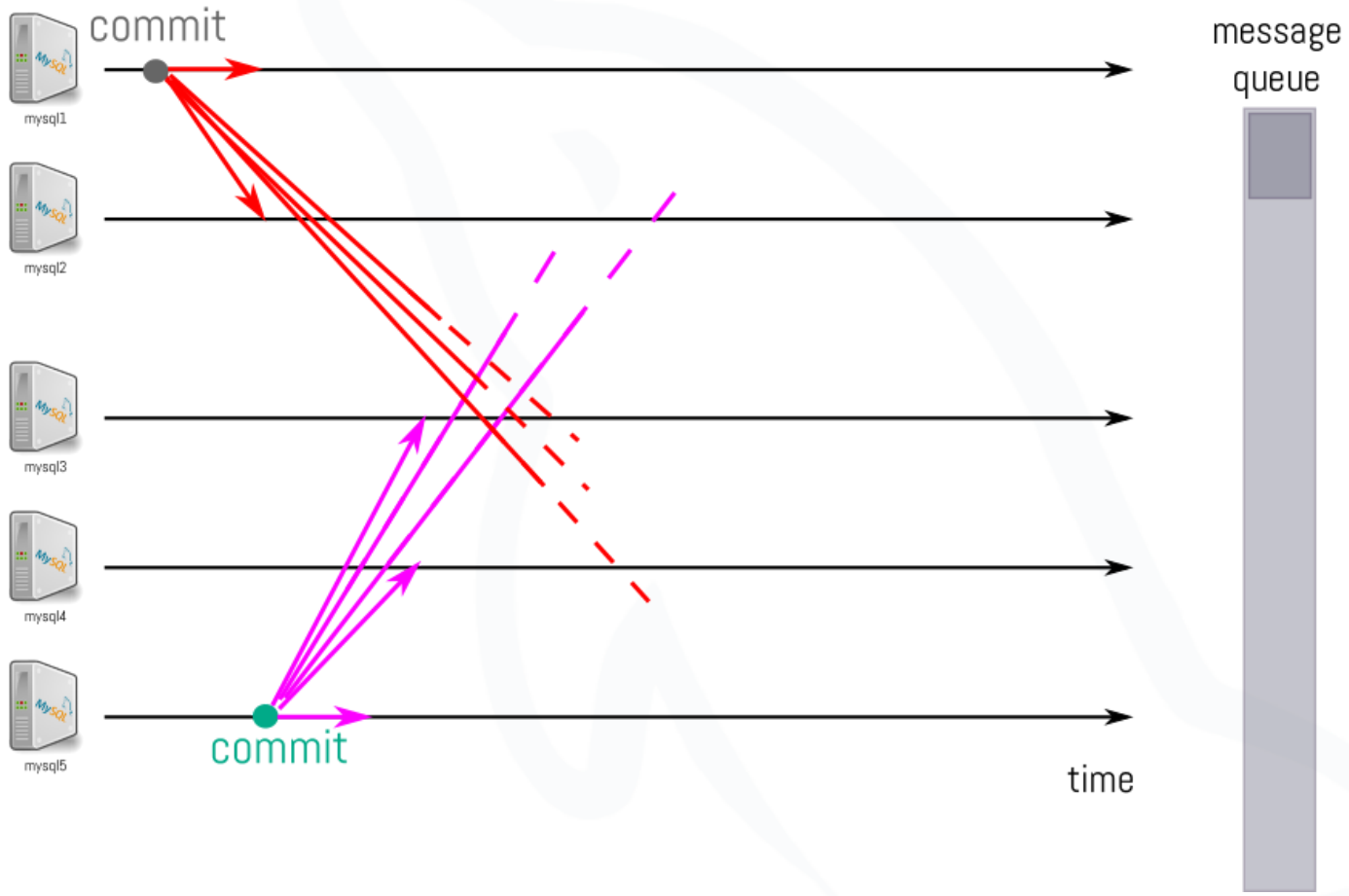
Group Replication : Total Order Delivery - GTID



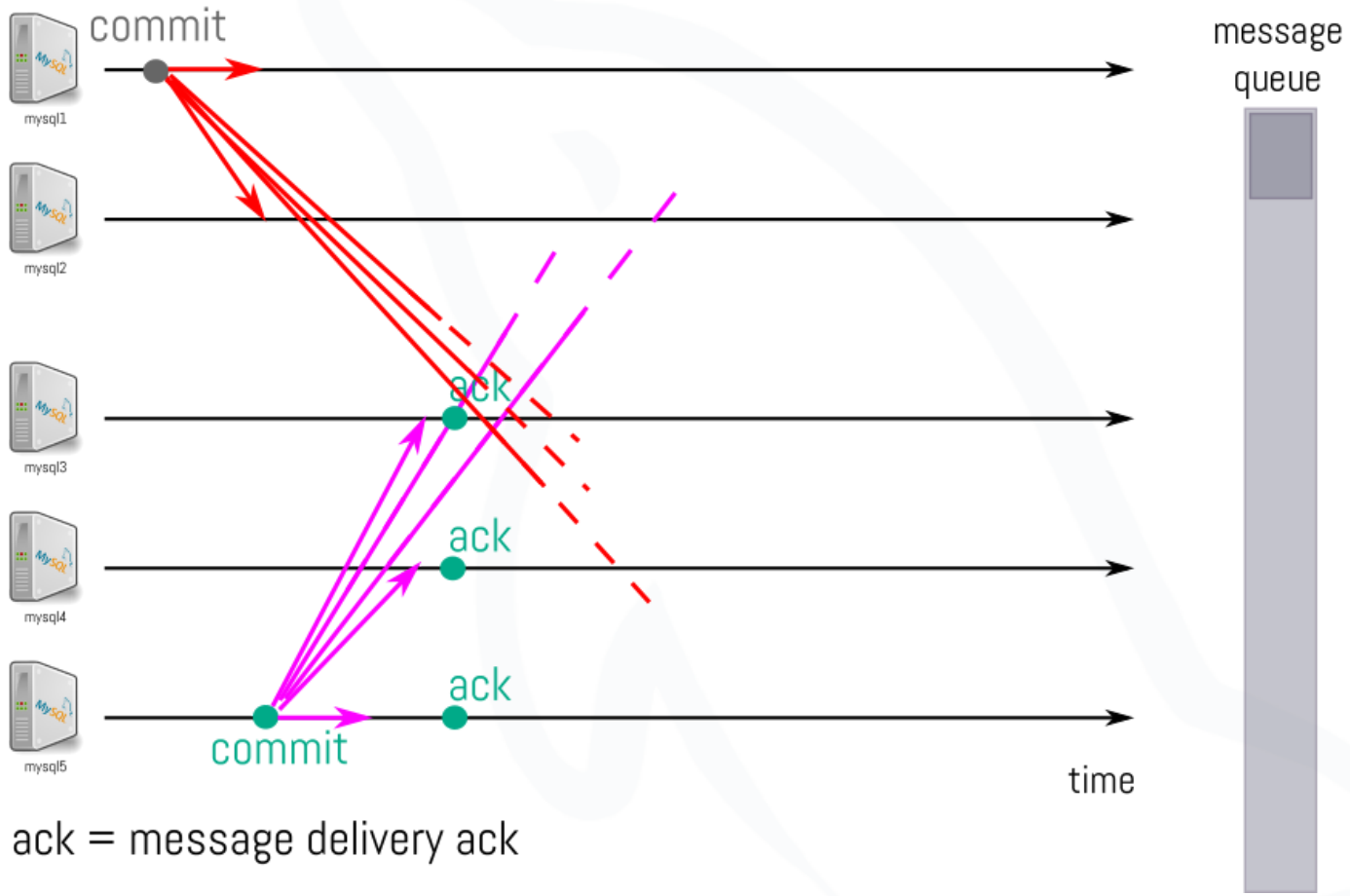
Group Replication : Total Order Delivery - GTID



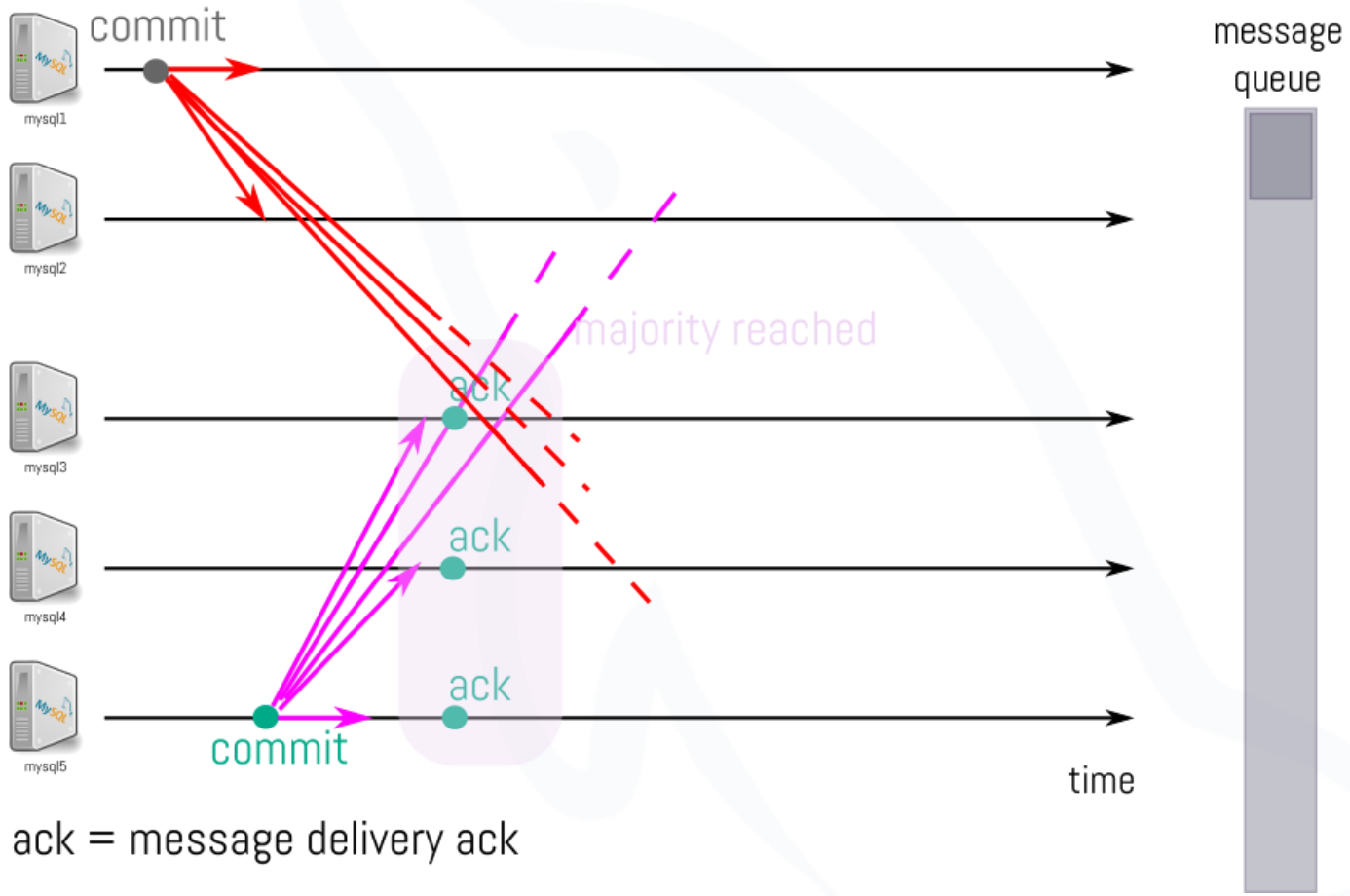
Group Replication : Total Order Delivery - GTID



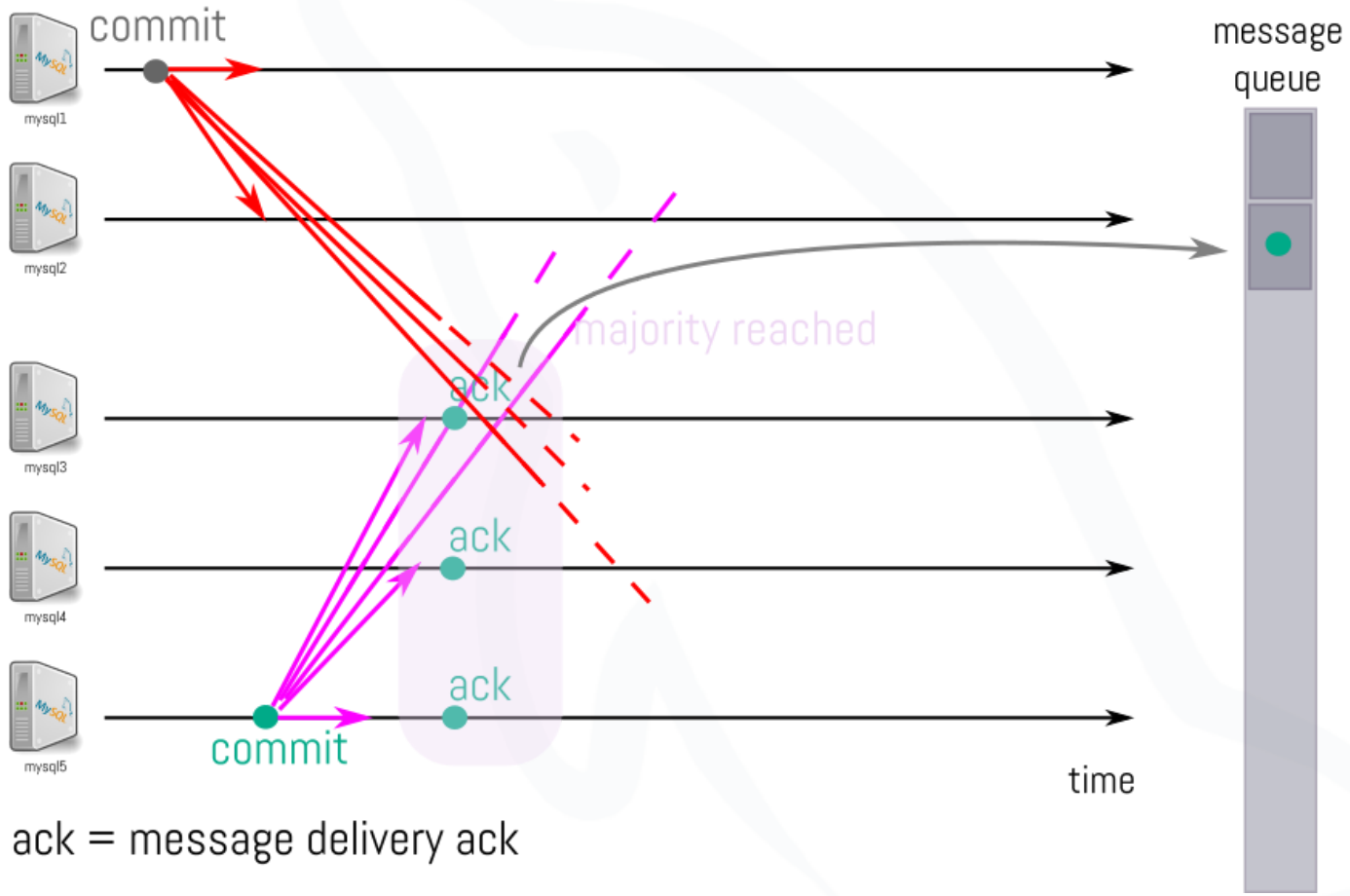
Group Replication : Total Order Delivery - GTID



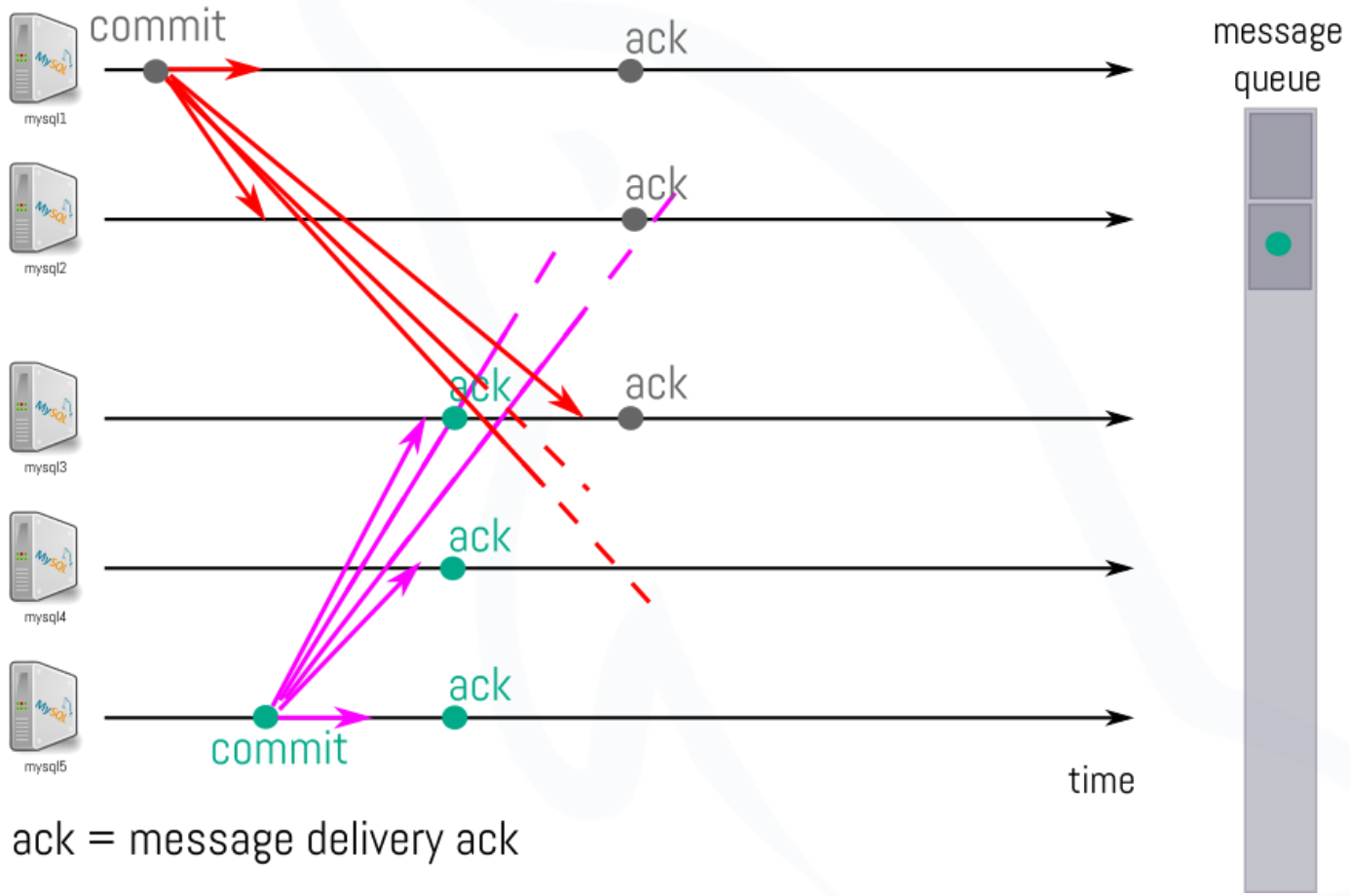
Group Replication : Total Order Delivery - GTID



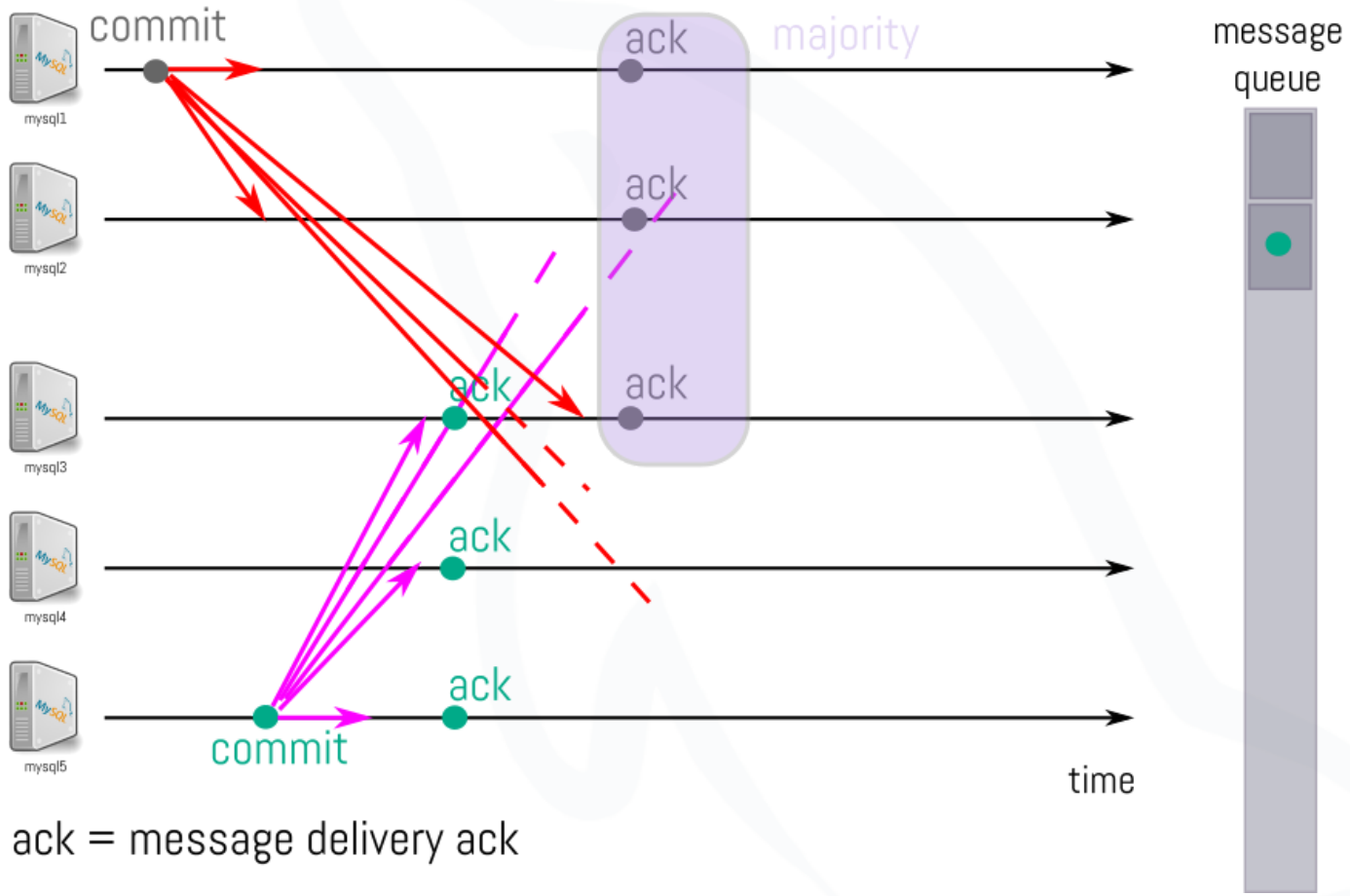
Group Replication : Total Order Delivery - GTID



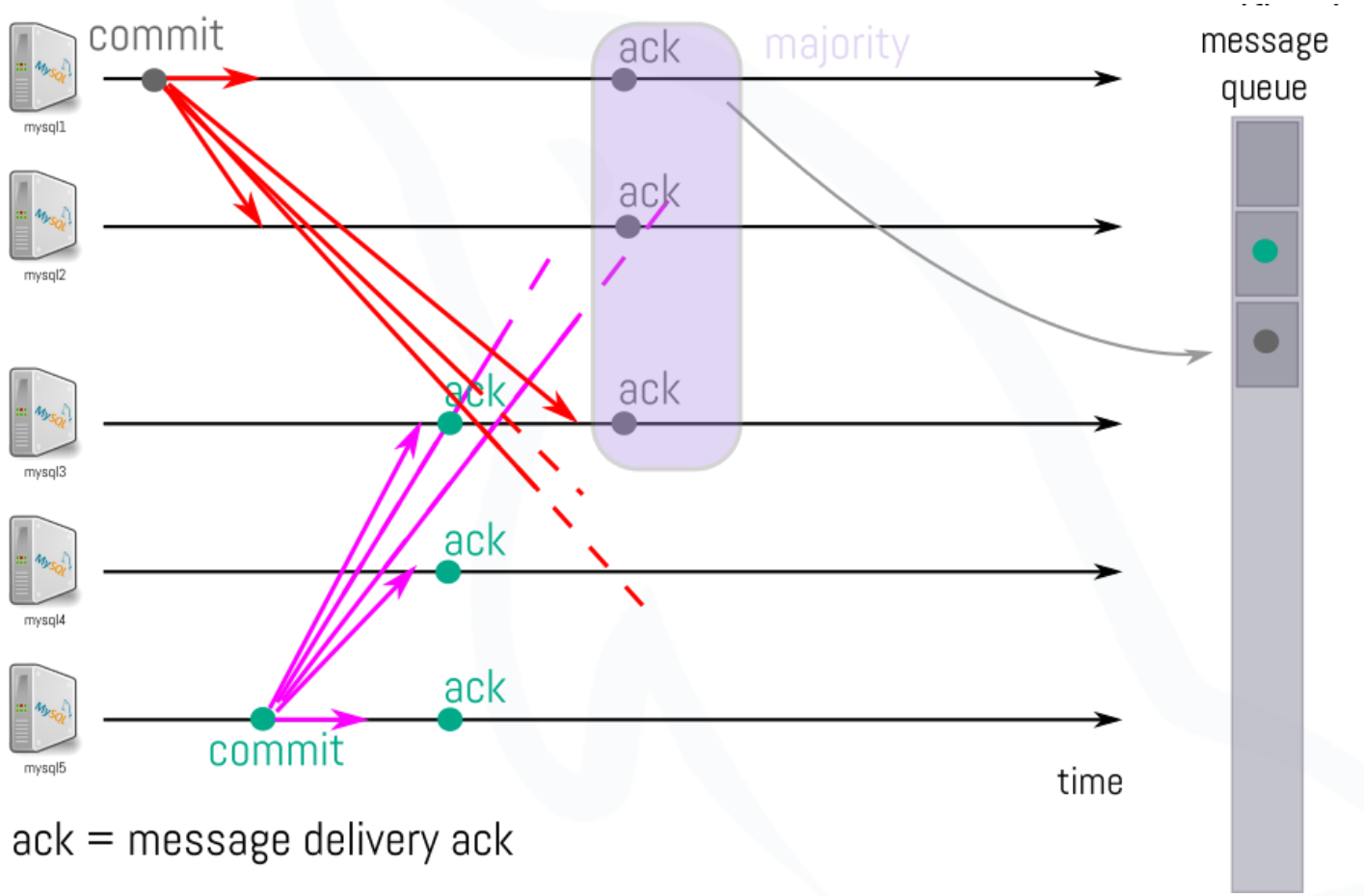
Group Replication : Total Order Delivery - GTID



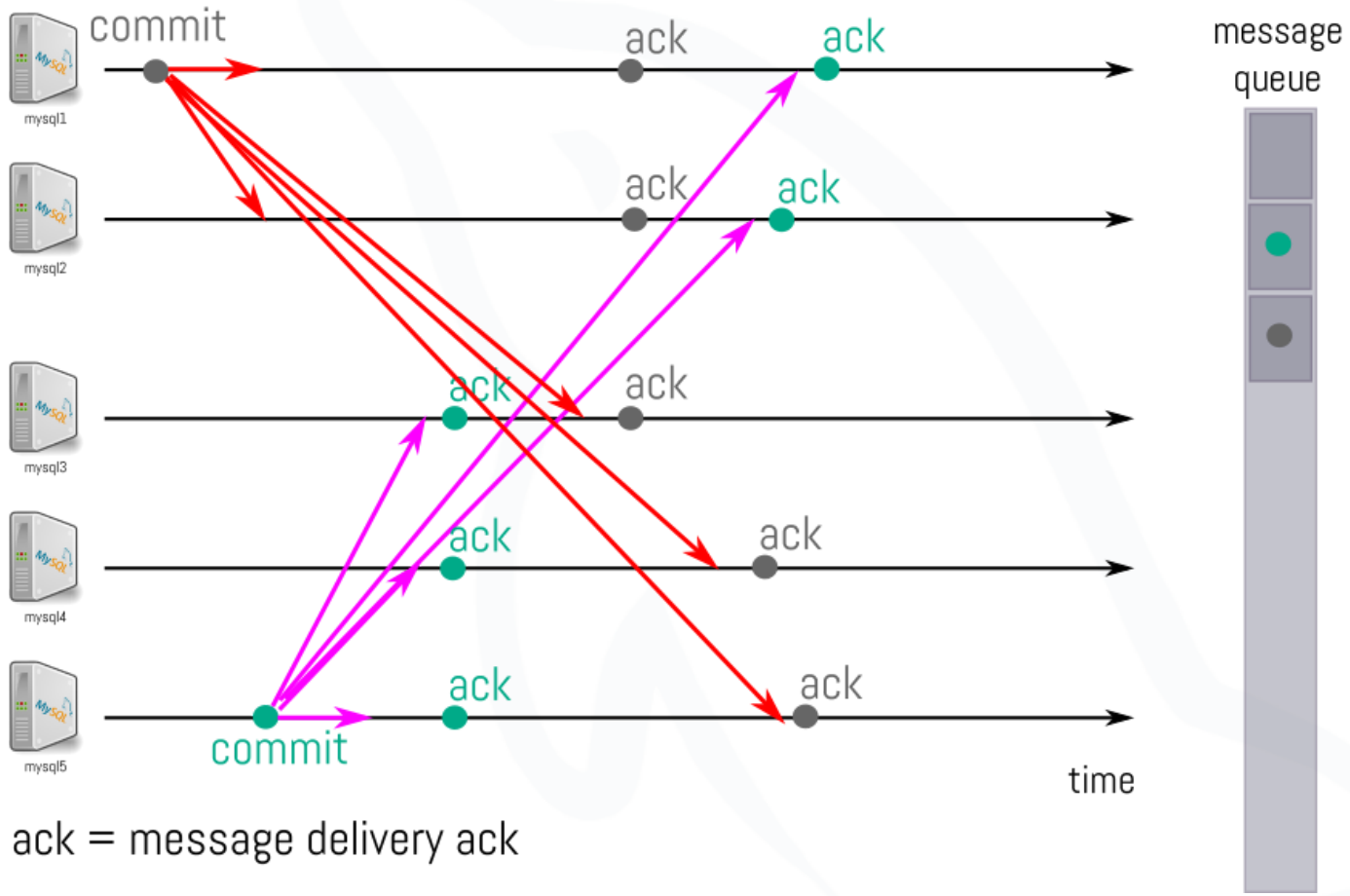
Group Replication : Total Order Delivery - GTID



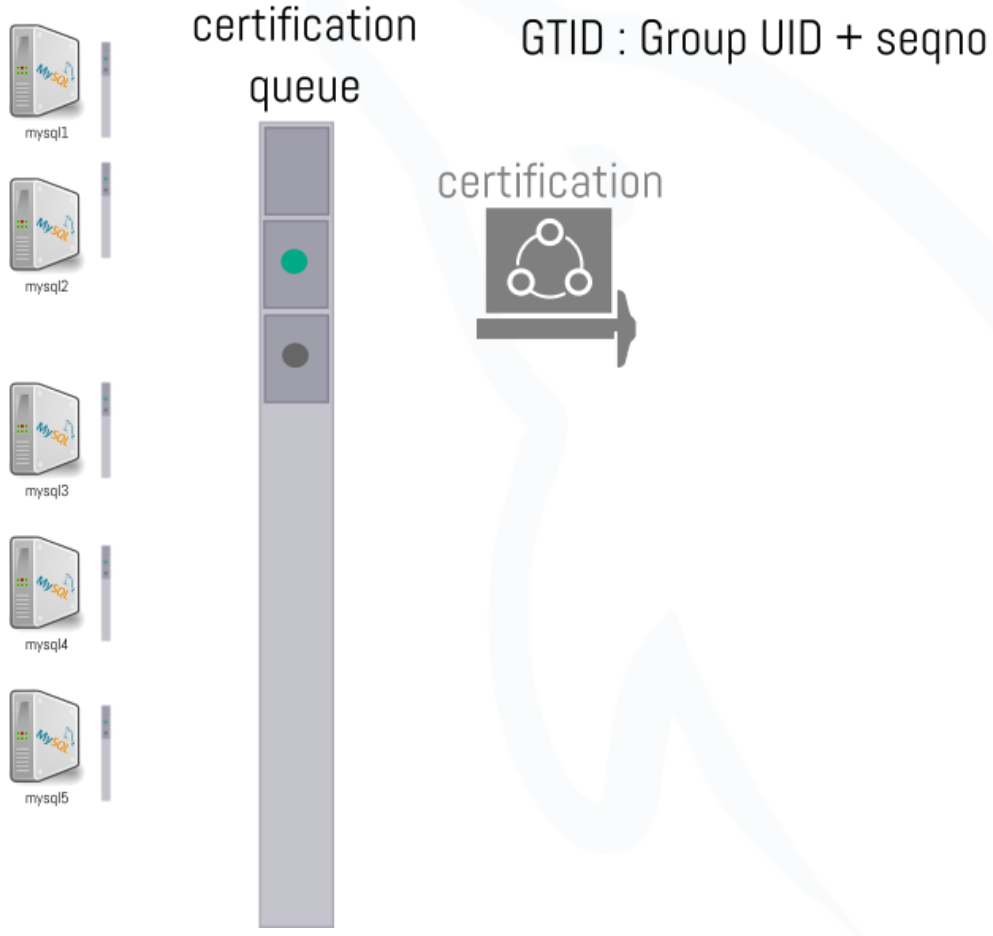
Group Replication : Total Order Delivery - GTID



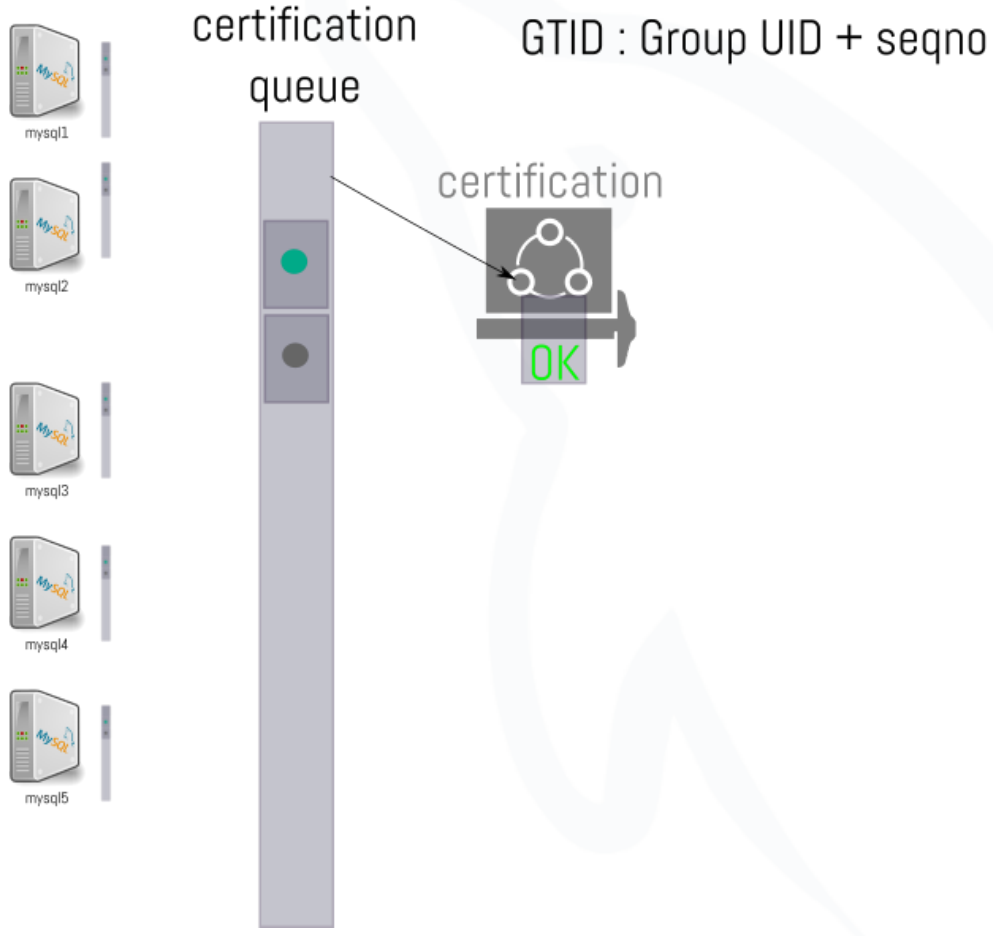
Group Replication : Total Order Delivery - GTID



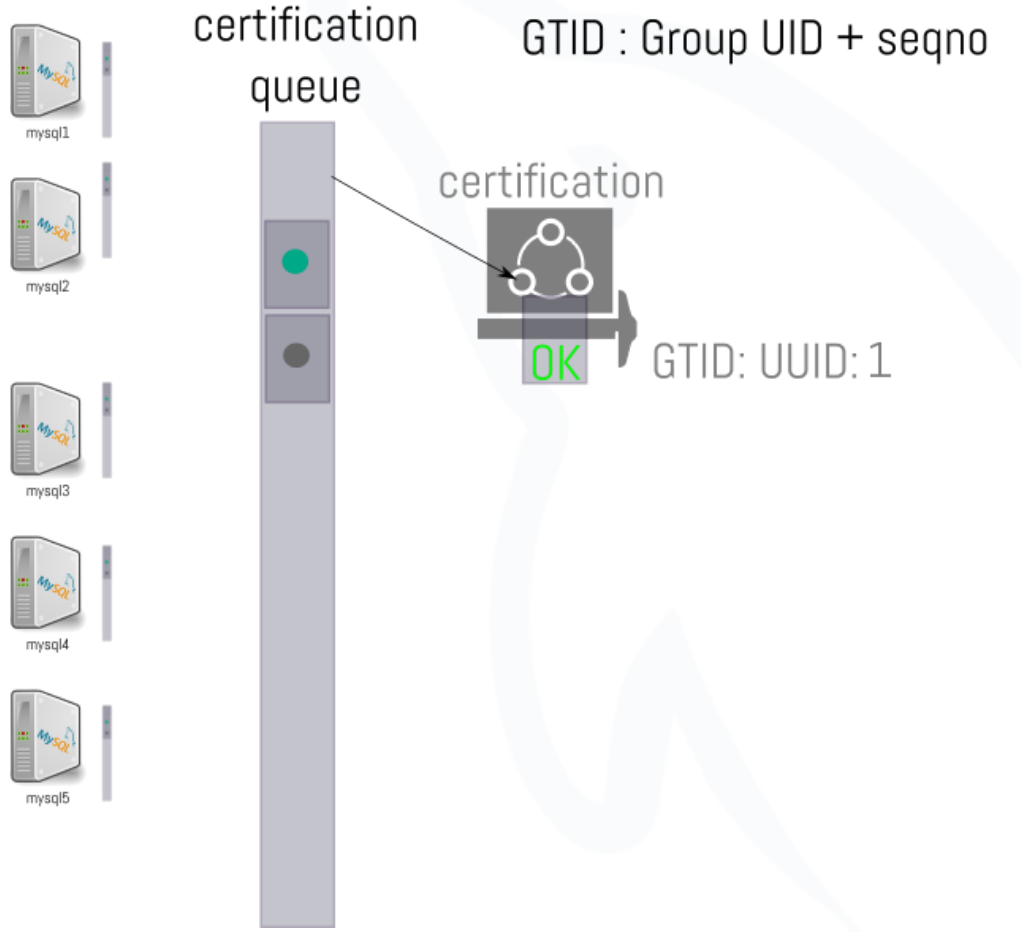
Group Replication : Total Order Delivery - GTID



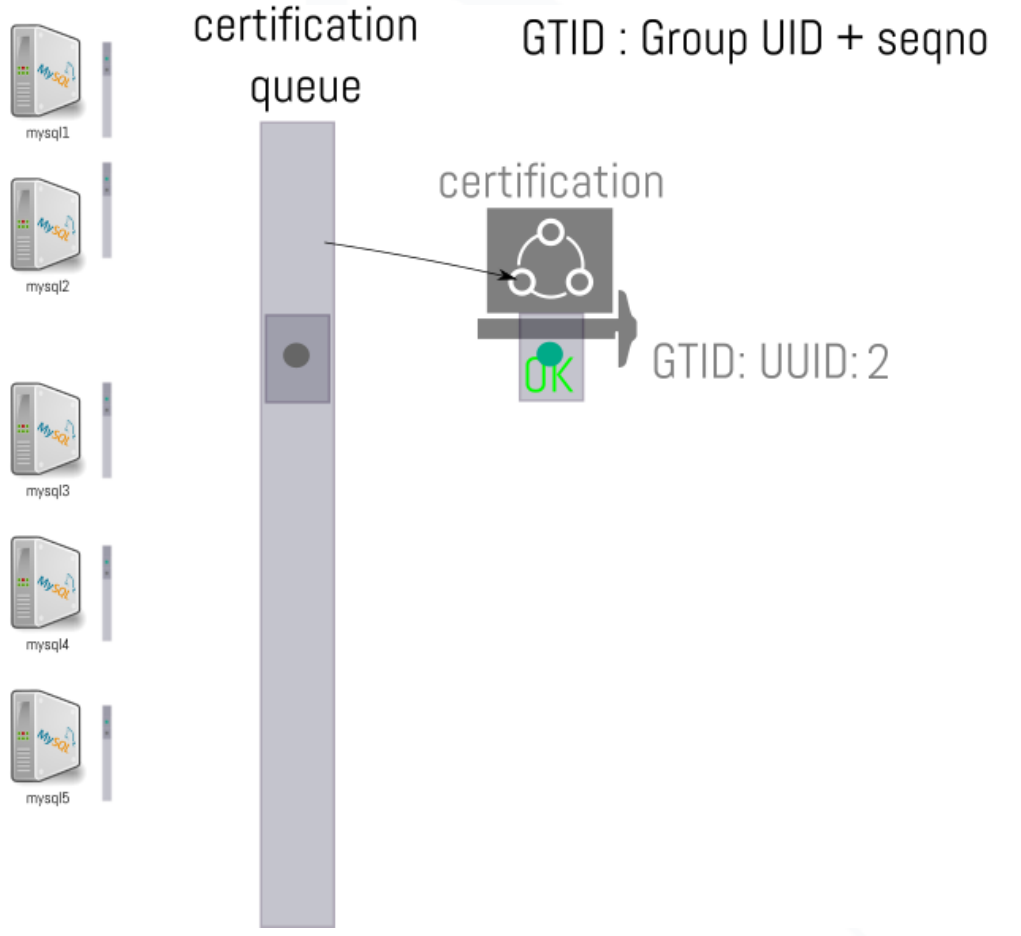
Group Replication : Total Order Delivery - GTID



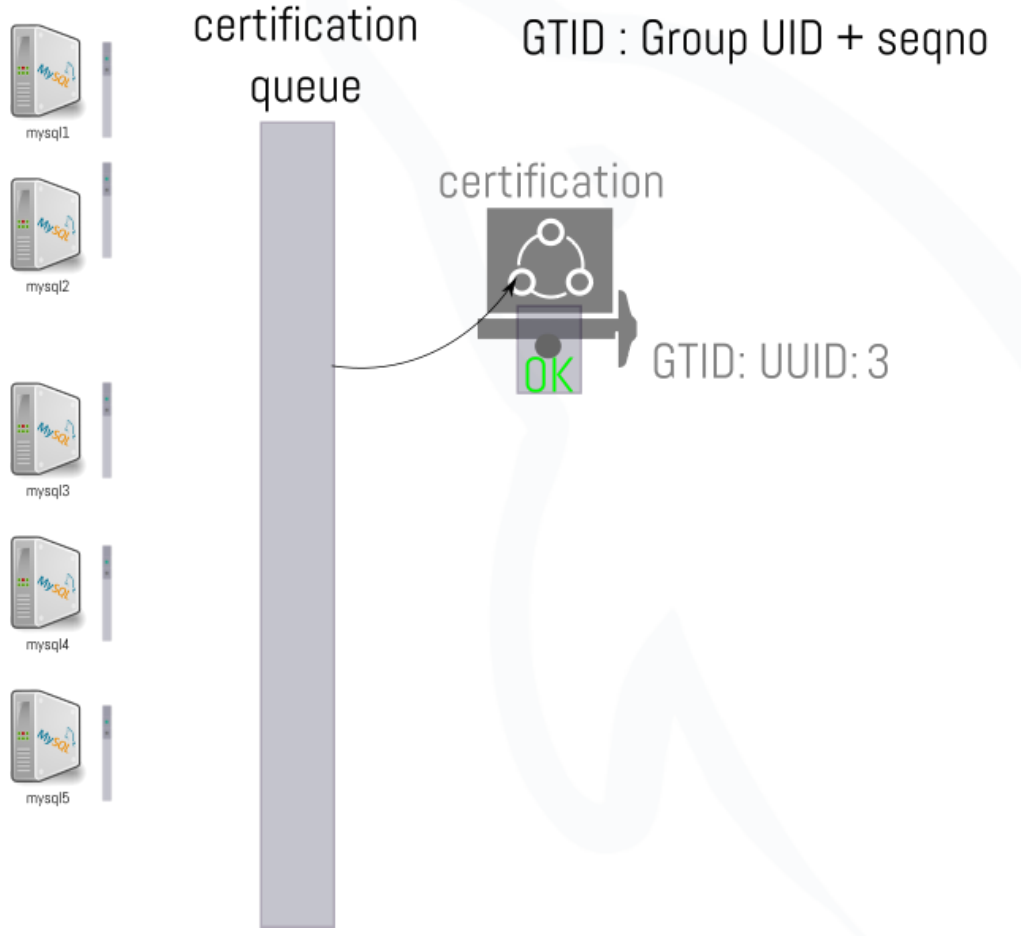
Group Replication : Total Order Delivery - GTID



Group Replication : Total Order Delivery - GTID



Group Replication : Total Order Delivery - GTID



Group Replication : GTID

The previous example was not totally in sync with reality. In fact, a writer allocates a block of GTID and when we have multiple writes (*multi-primary mode*) all writers will use GTID sequence numbers in their allocated block.

The size of the block is defined by `group_replication_gtid_assignment_block_size` (default to 1M)

Group Replication : GTID

Example:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355
```

Group Replication : GTID

Example:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355
```

New write on an other node:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355,1000354
```

Group Replication : GTID

Example:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355
```

New write on an other node:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355,1000354
```

Let's write on the third node:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355:1000354:2000354
```

Group Replication : GTID

Example:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355
```

New write on an other node:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355,1000354
```

Let's write on the third node:

```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-355:1000354:2000354
```

And writing back on the first one:

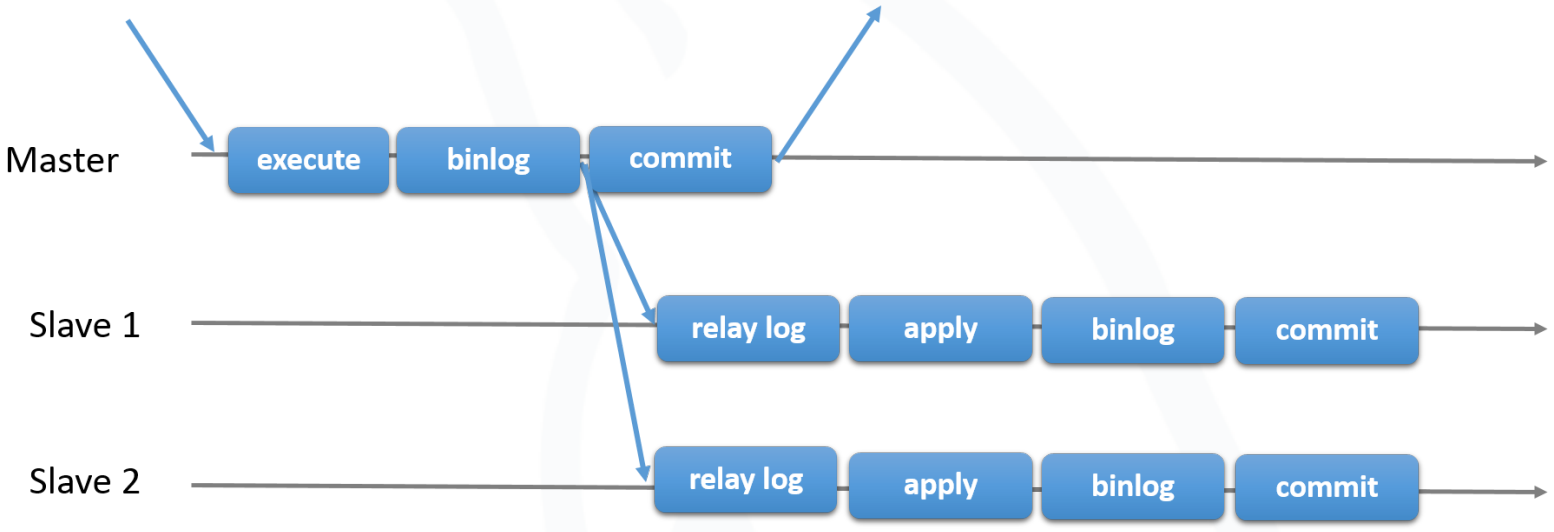
```
Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-08002718d305:1-356:1000354:2000354
```

done !

Return from Commit

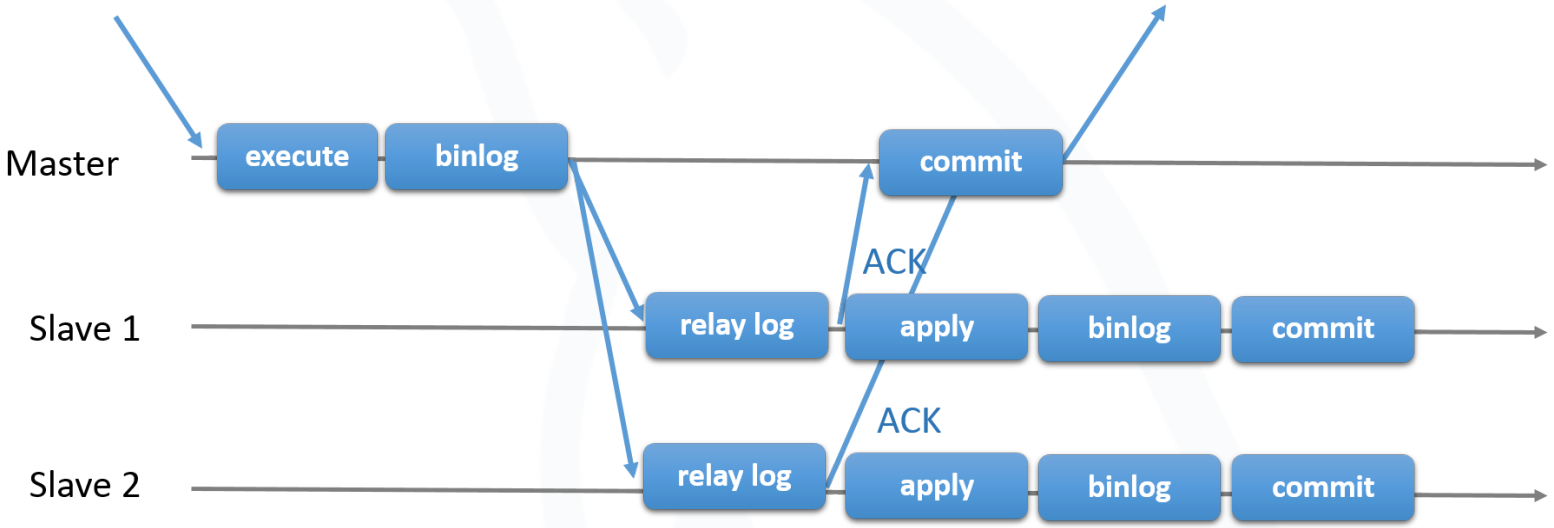
Group Replication: return from commit

Asynchronous Replication:



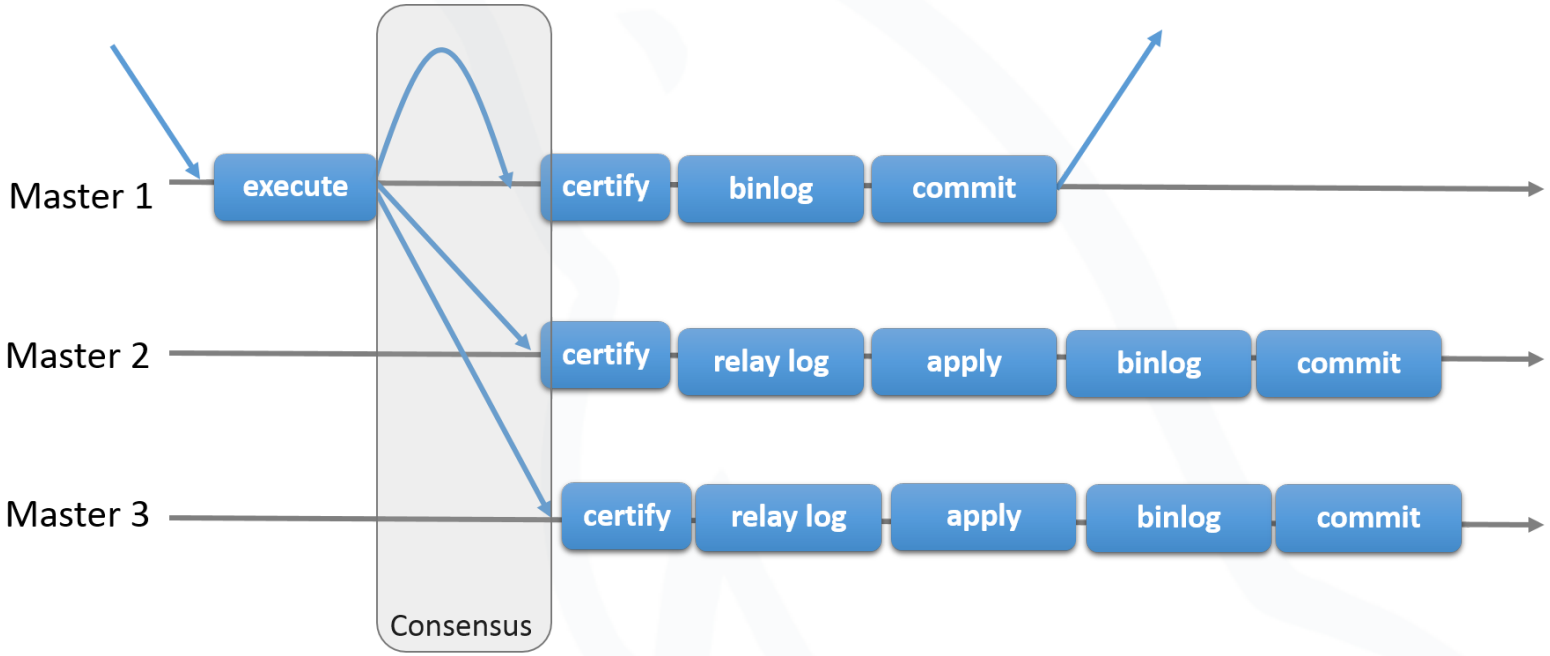
Group Replication: return from commit (2)

Semi-Sync Replication:



Group Replication: return from commit (3)

Group Replication:



Does this mean we can have a distant node and always let it ack later ?

Does this mean we can have a distant node and always let it ack later ?

NO!

Does this mean we can have a distant node and always let it ack later ?

NO!

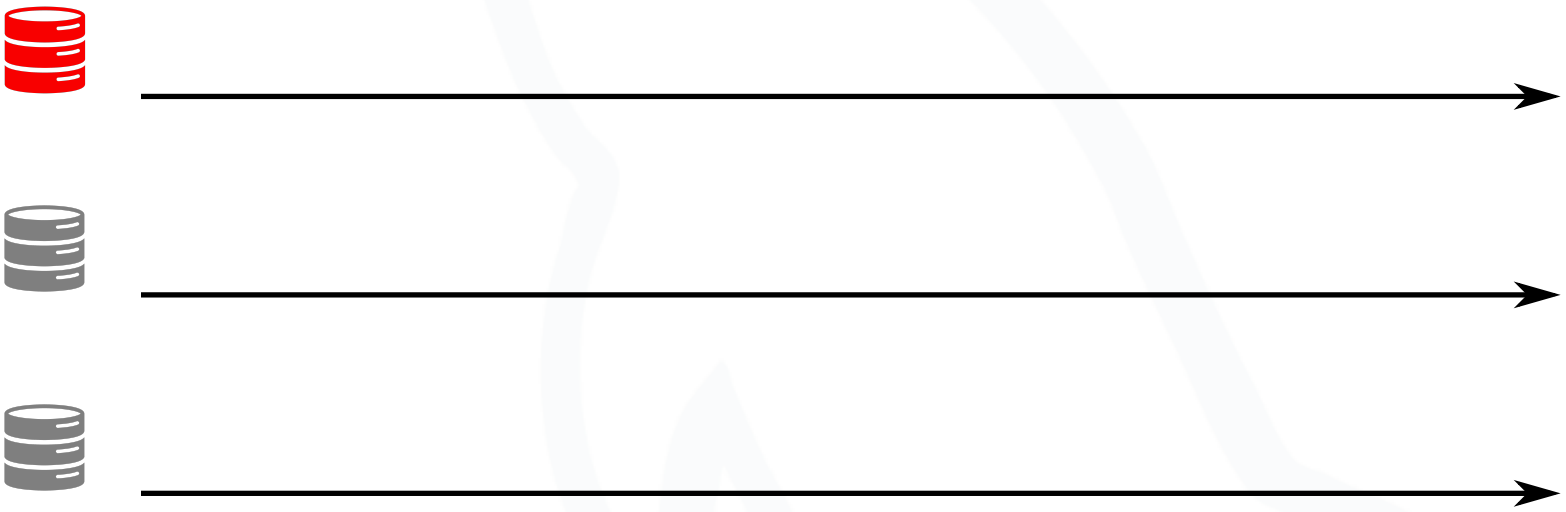
Because the system has to wait for the noop (single skip message) from the “distant” node where latency is higher

The size of the GCS consensus messages window can be get and set from UDF functions: *group_replication_get_write_concurrency()*, *group_replication_set_write_concurrency()*

```
mysql> select group_replication_get_write_concurrency();
+-----------------------------------------+
| group_replication_get_write_concurrency() |
+-----------------------------------------+
|                                      10 |
+-----------------------------------------+
```

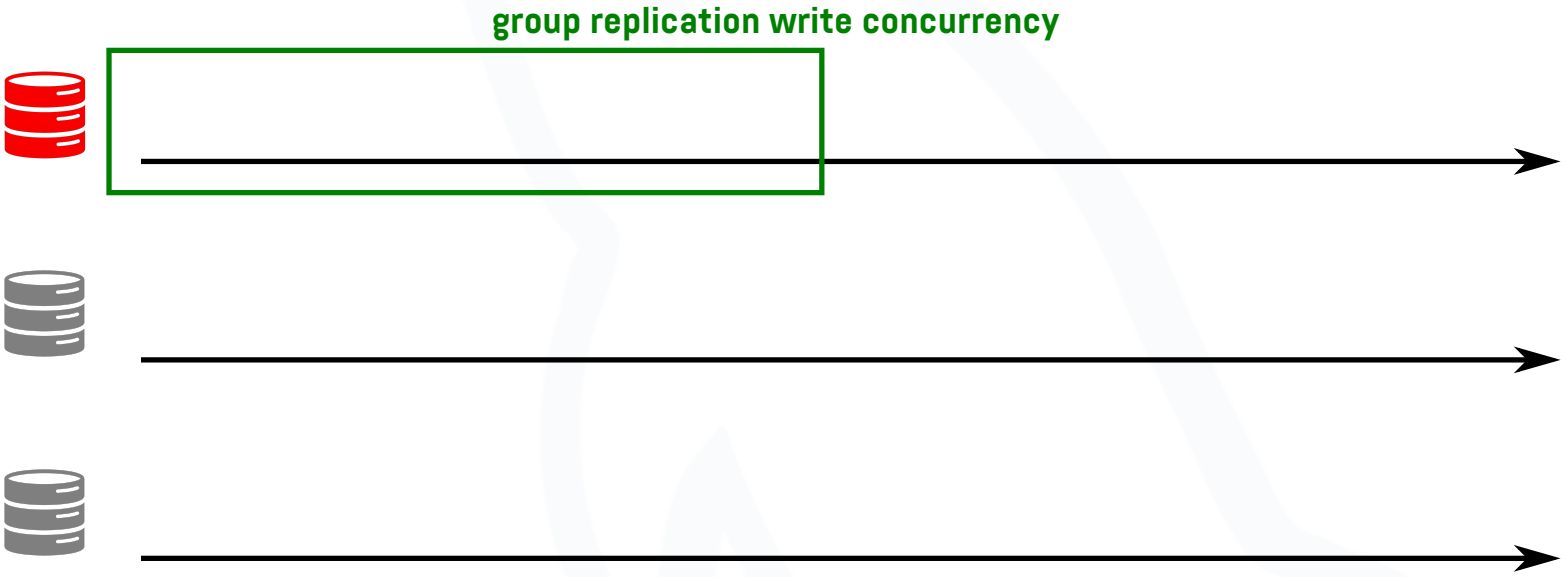
Event Horizon

GCS Write Consensus Concurrency



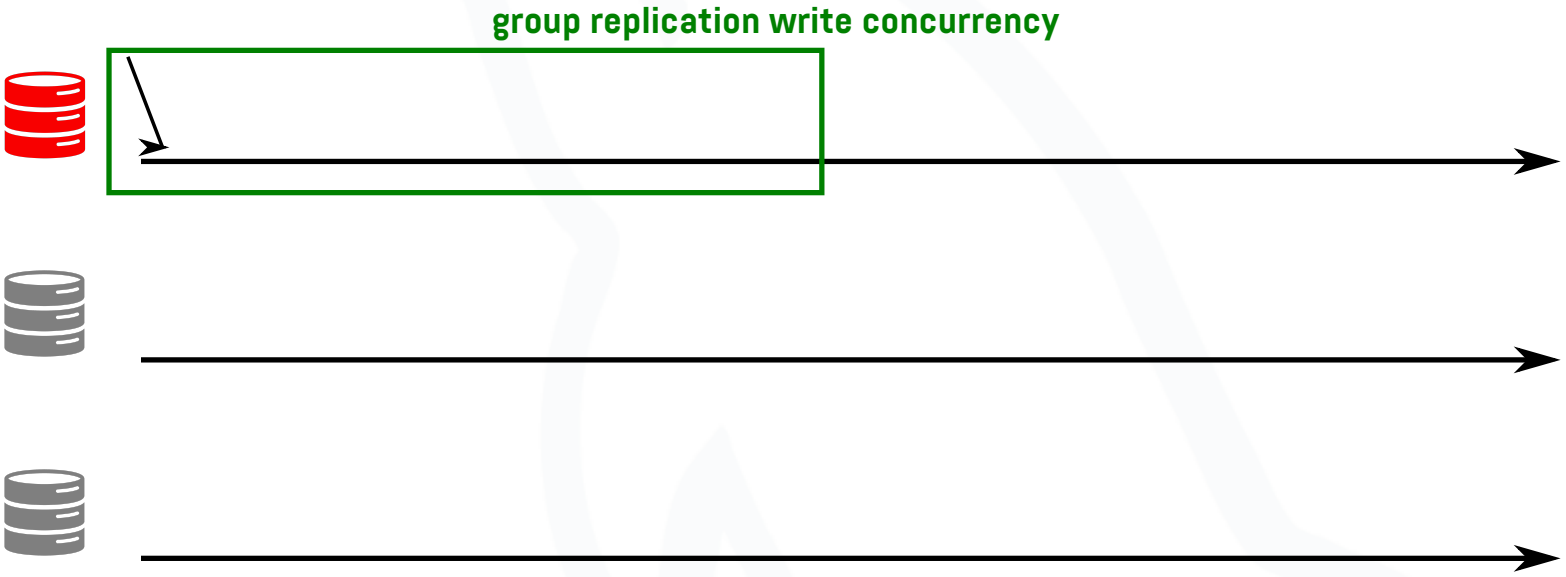
Event Horizon

GCS Write Consensus Concurrency



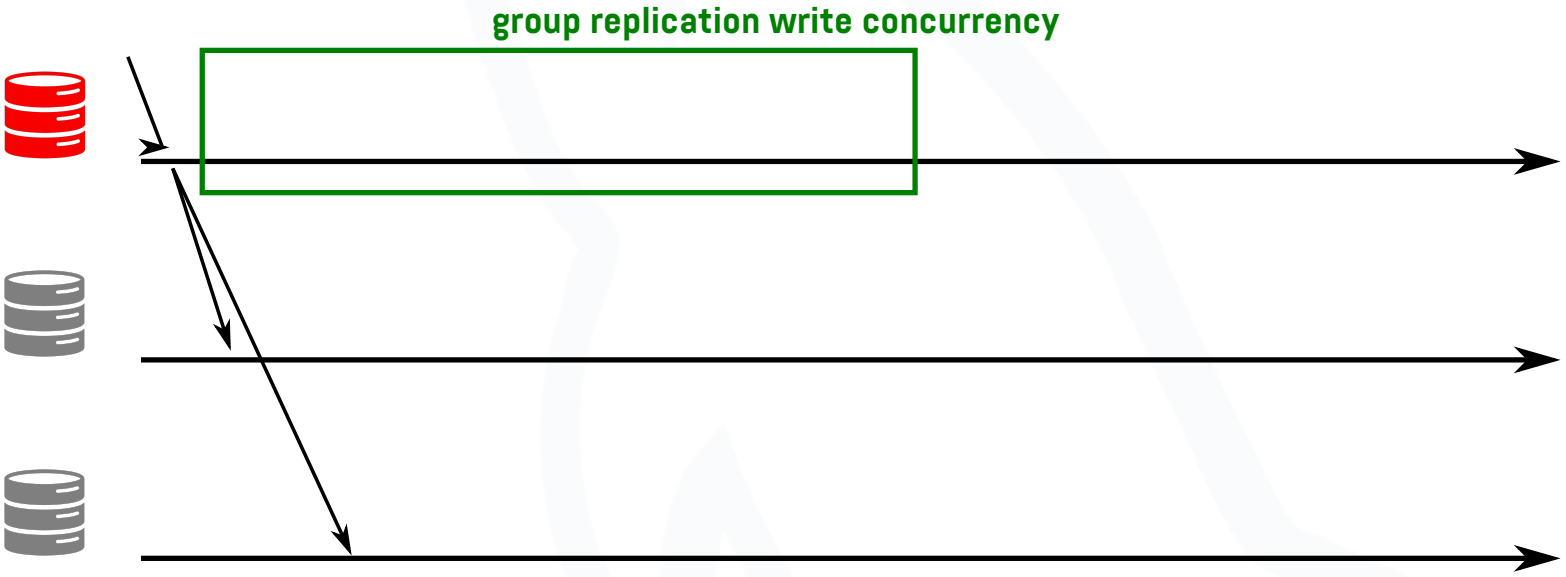
Event Horizon

GCS Write Consensus Concurrency



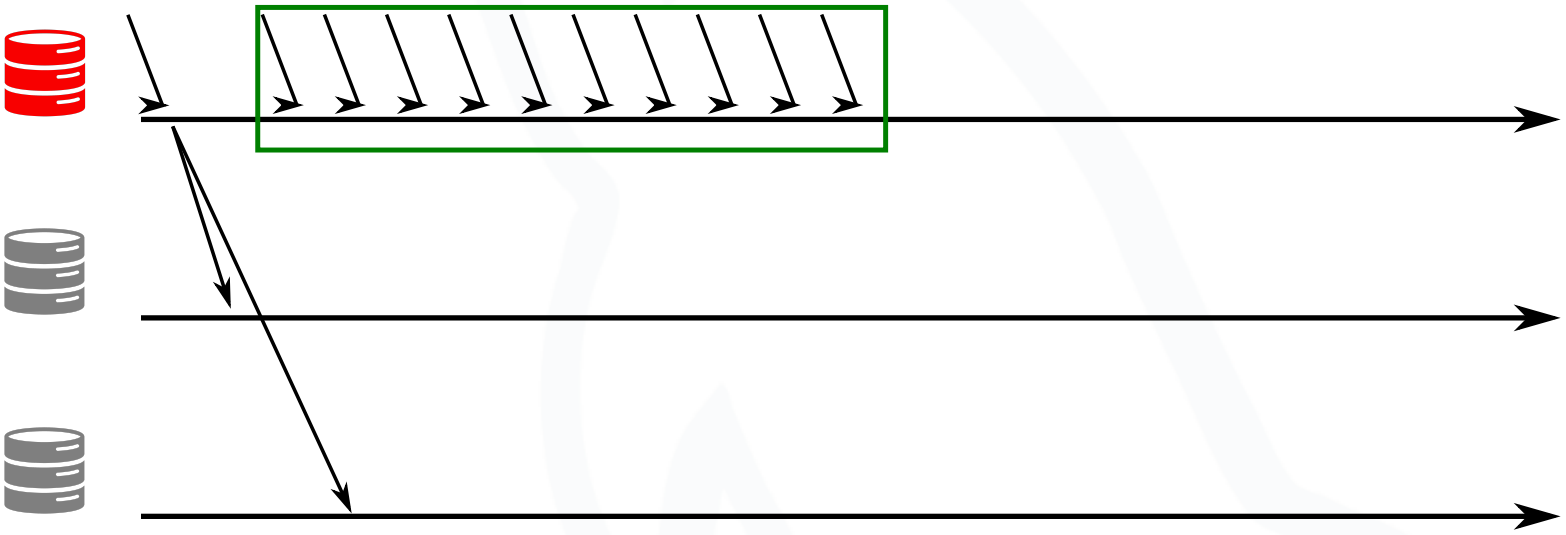
Event Horizon

GCS Write Consensus Concurrency



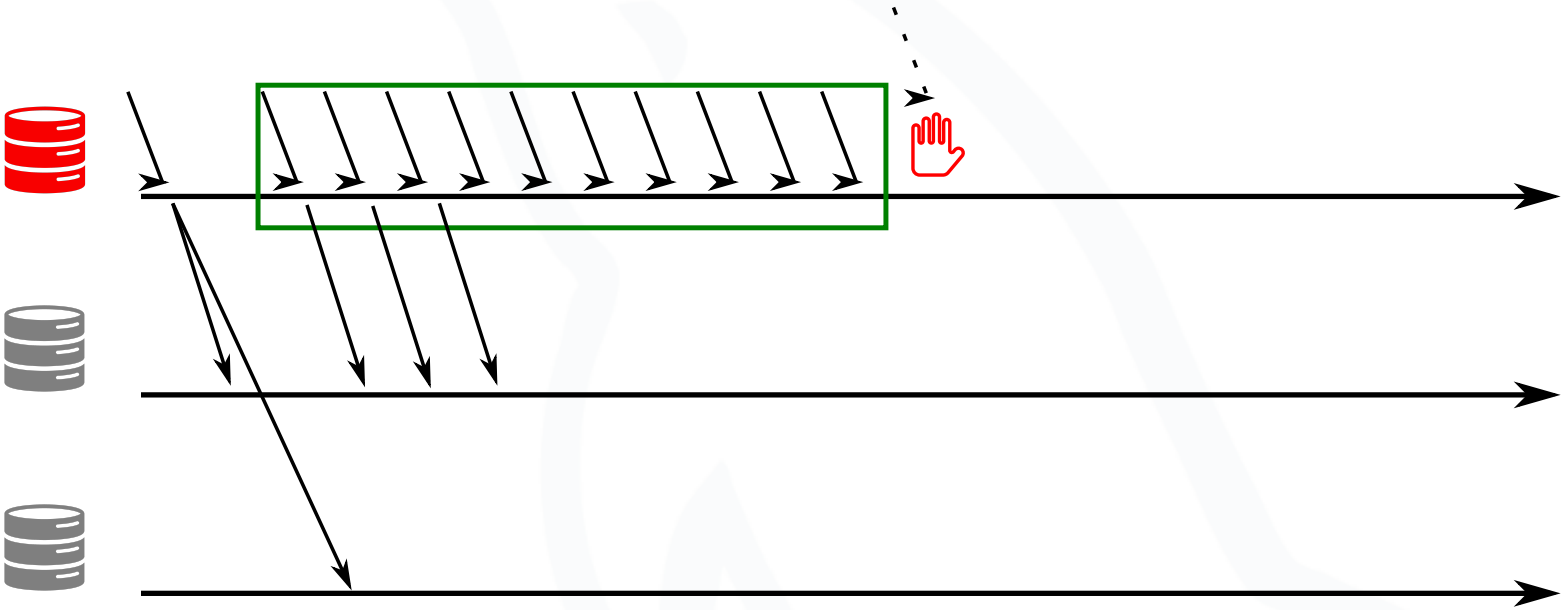
Event Horizon

GCS Write Consensus Concurrency



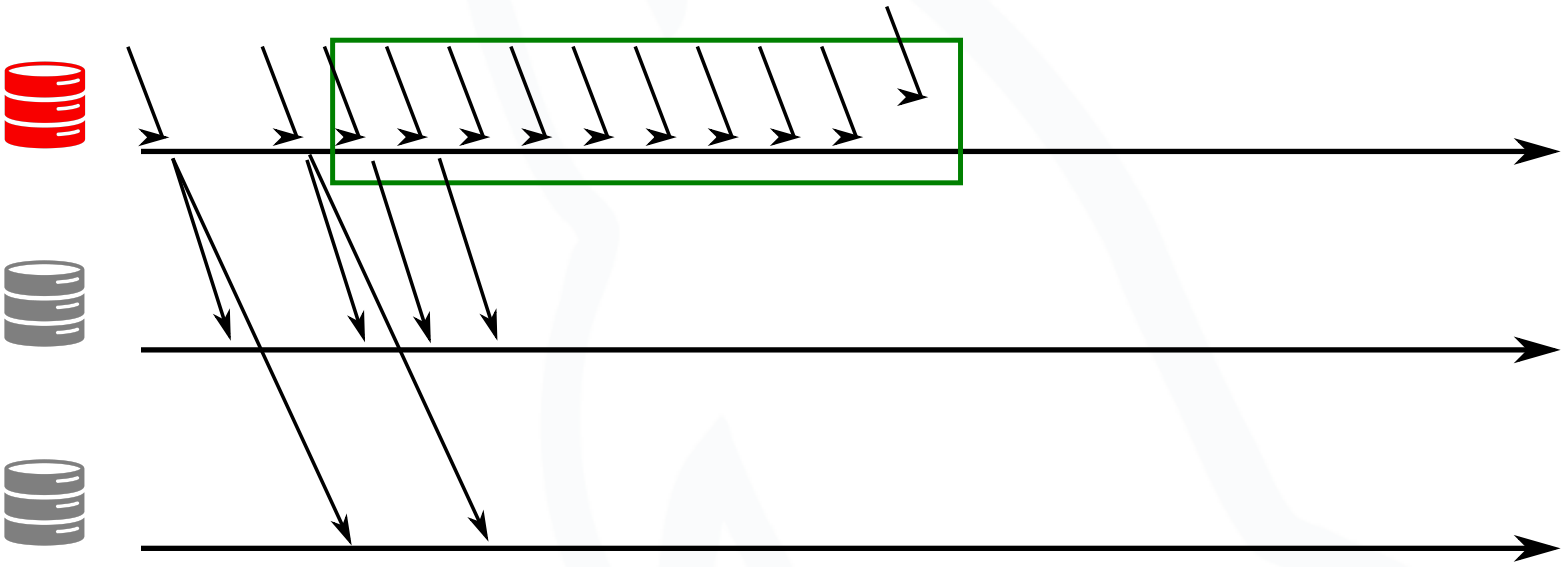
Event Horizon

GCS Write Consensus Concurrency



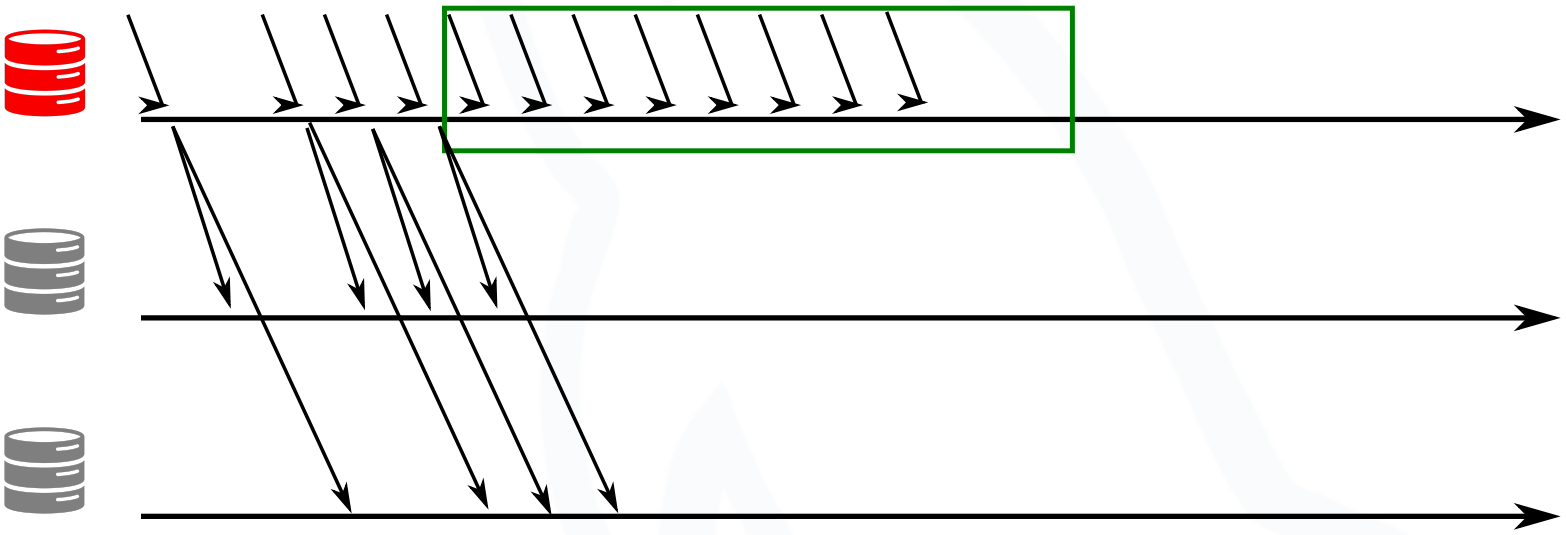
Event Horizon

GCS Write Consensus Concurrency



Event Horizon

GCS Write Consensus Concurrency



conflict

Optimistic Locking

Group Replication : Optimistic Locking

Group Replication uses optimistic locking

Group Replication : Optimistic Locking

Group Replication uses optimistic locking

- during a transaction, **local (InnoDB) locking** happens

Group Replication : Optimistic Locking

Group Replication uses optimistic locking

- during a transaction, **local (InnoDB) locking** happens
- **optimistically assumes** there will be no conflicts across nodes
  (no communication between nodes necessary)

Group Replication : Optimistic Locking

Group Replication uses optimistic locking

- during a transaction, **local (InnoDB) locking** happens
- **optimistically assumes** there will be no conflicts across nodes
  (no communication between nodes necessary)
- cluster-wide conflict resolution happens only at `COMMIT`, during **certification**

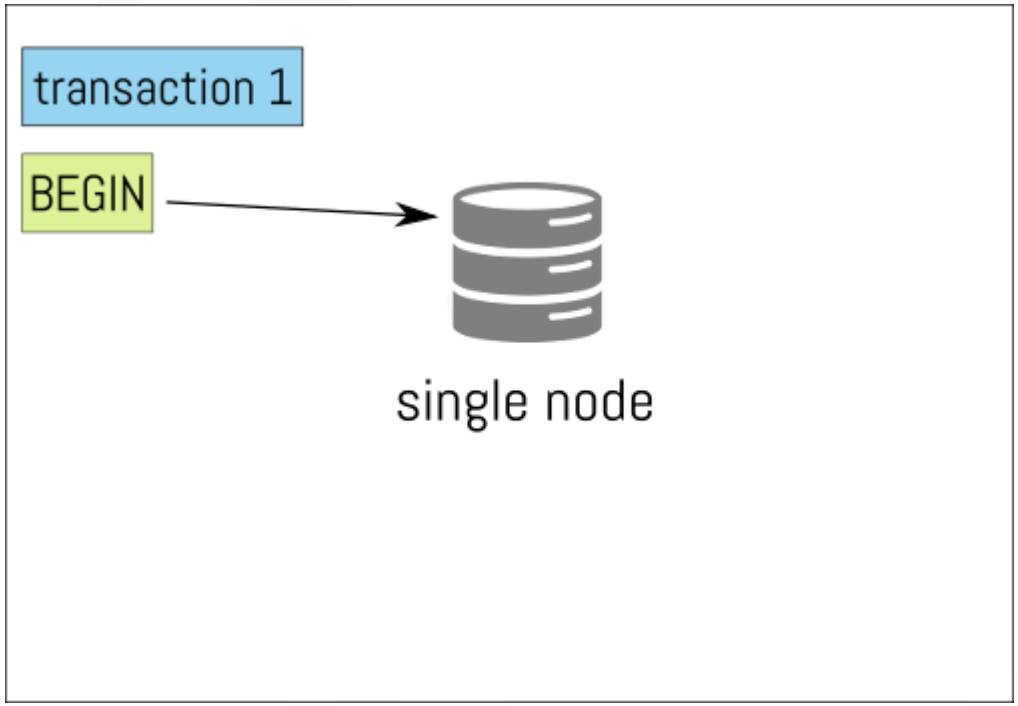
Group Replication : Optimistic Locking

Group Replication uses optimistic locking

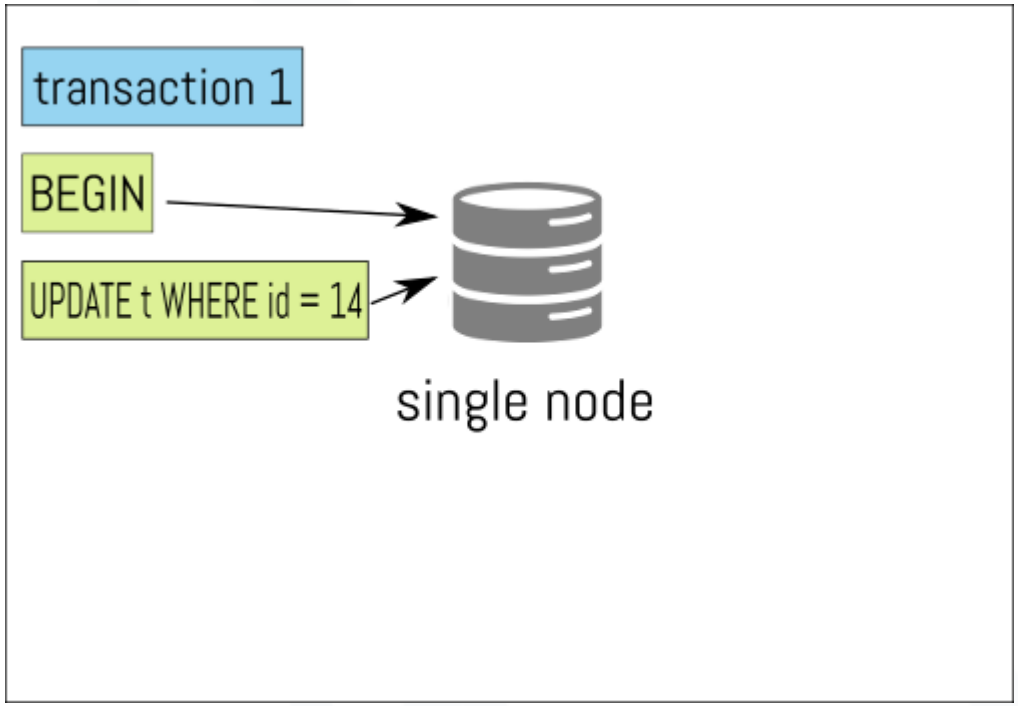
- during a transaction, **local (InnoDB) locking** happens
- **optimistically assumes** there will be no conflicts across nodes
  (no communication between nodes necessary)
- cluster-wide conflict resolution happens only at `COMMIT`, during **certification**

Let´s first have a look at the traditional locking to compare.

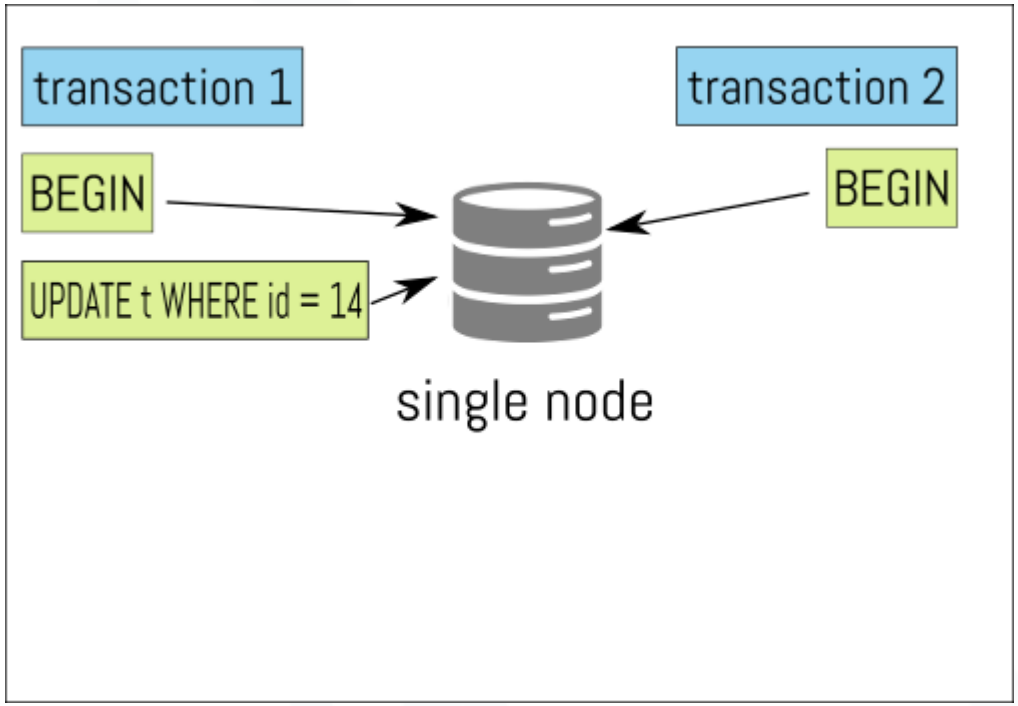
Traditional locking



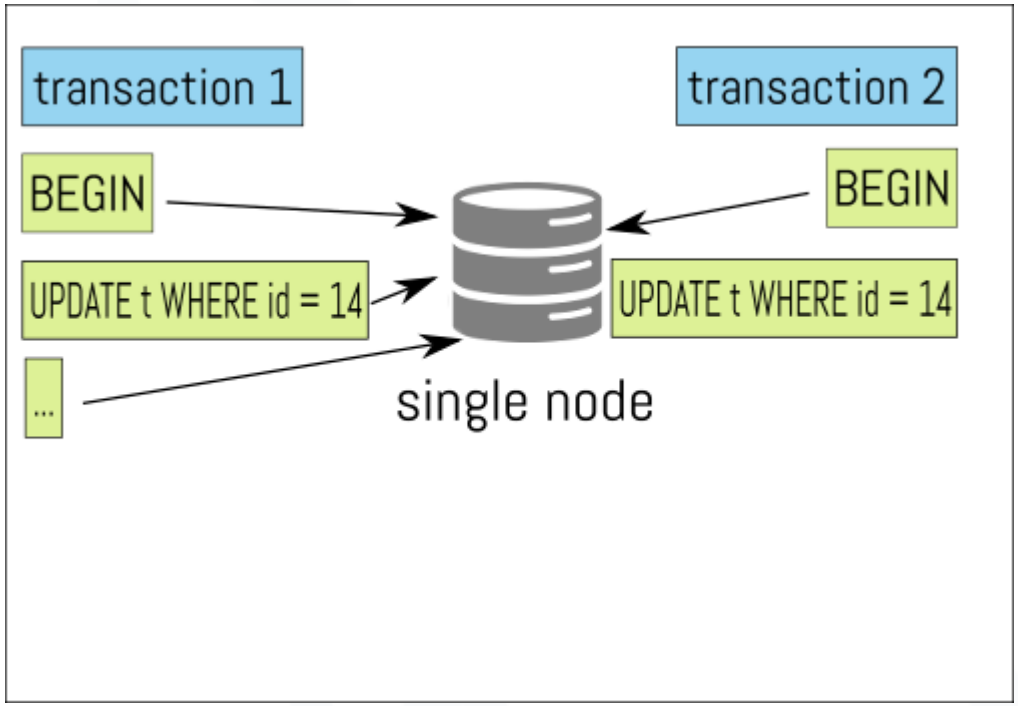
Traditional locking



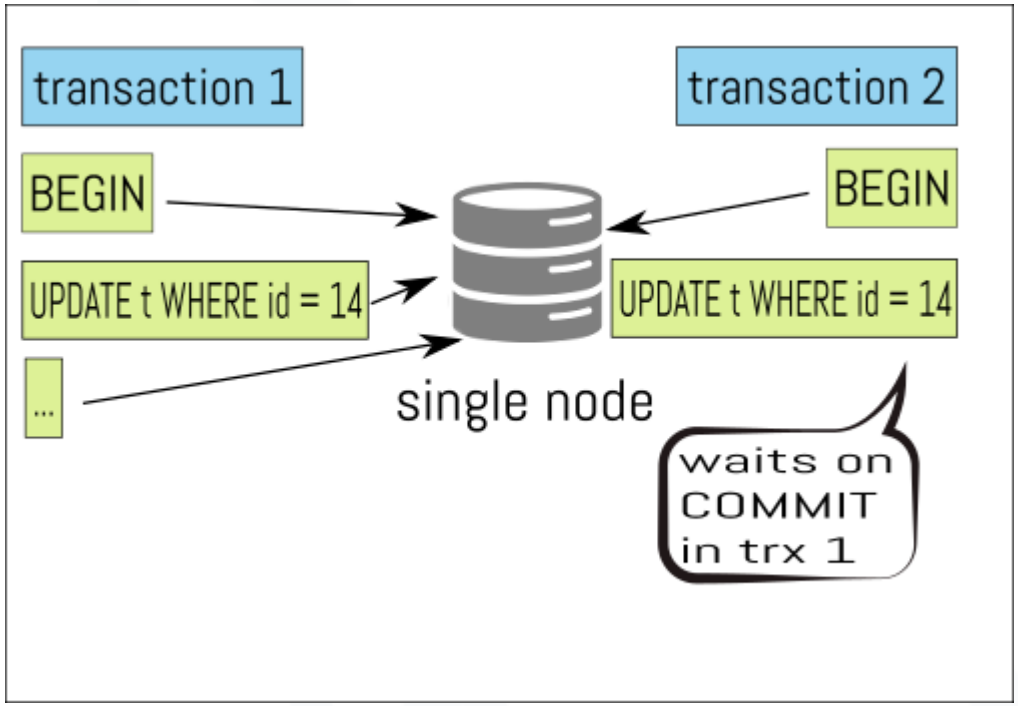
Traditional locking



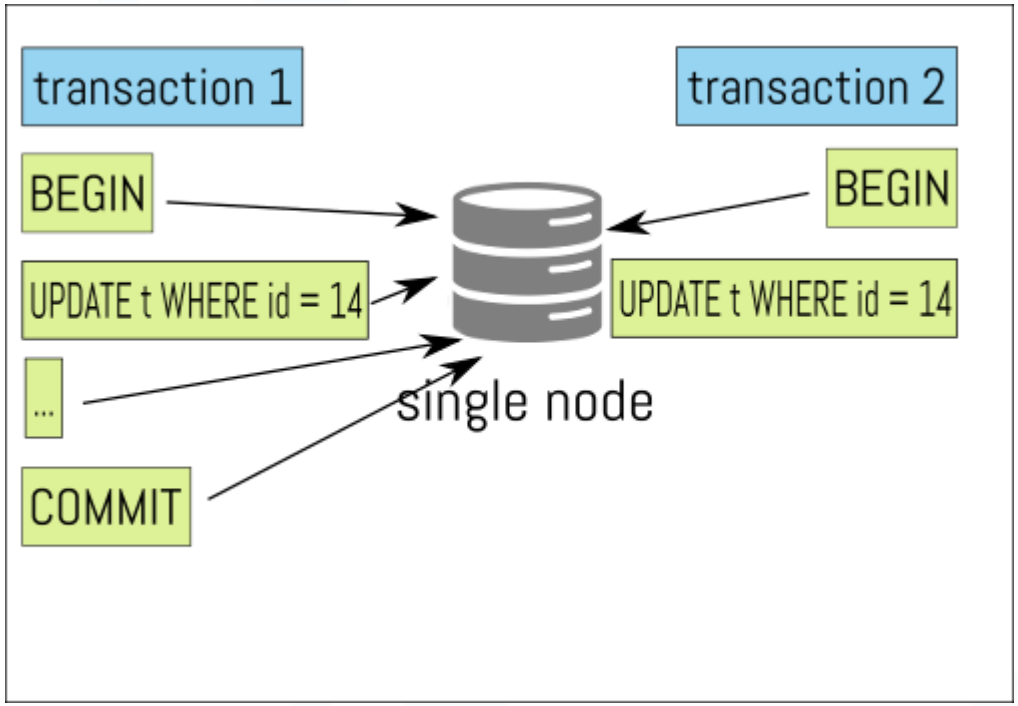
Traditional locking



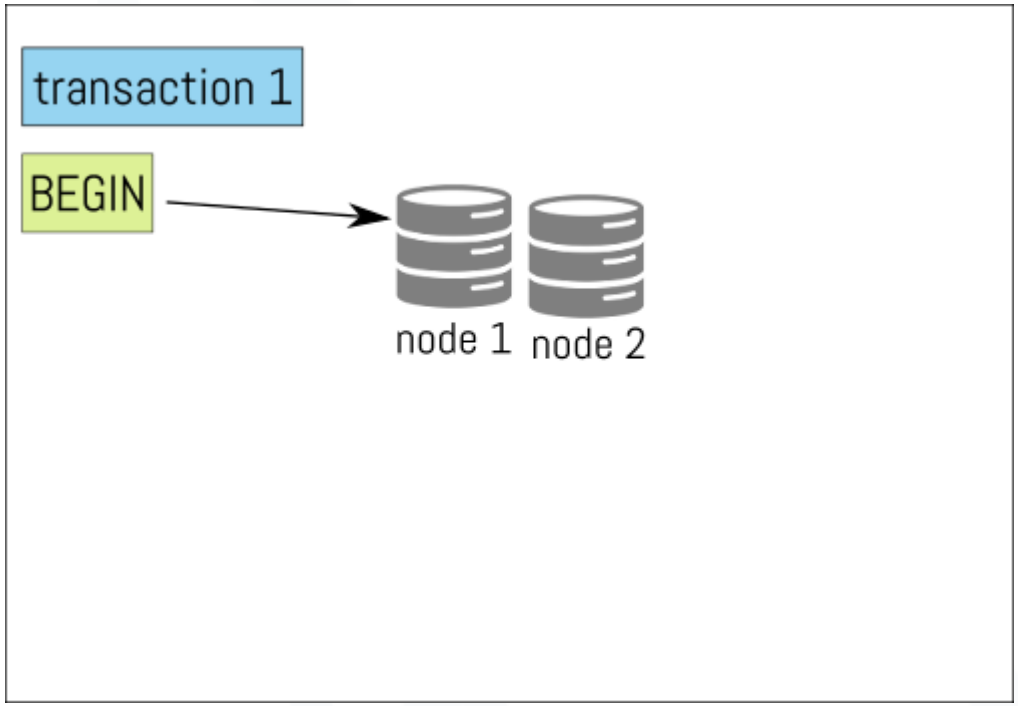
Traditional locking



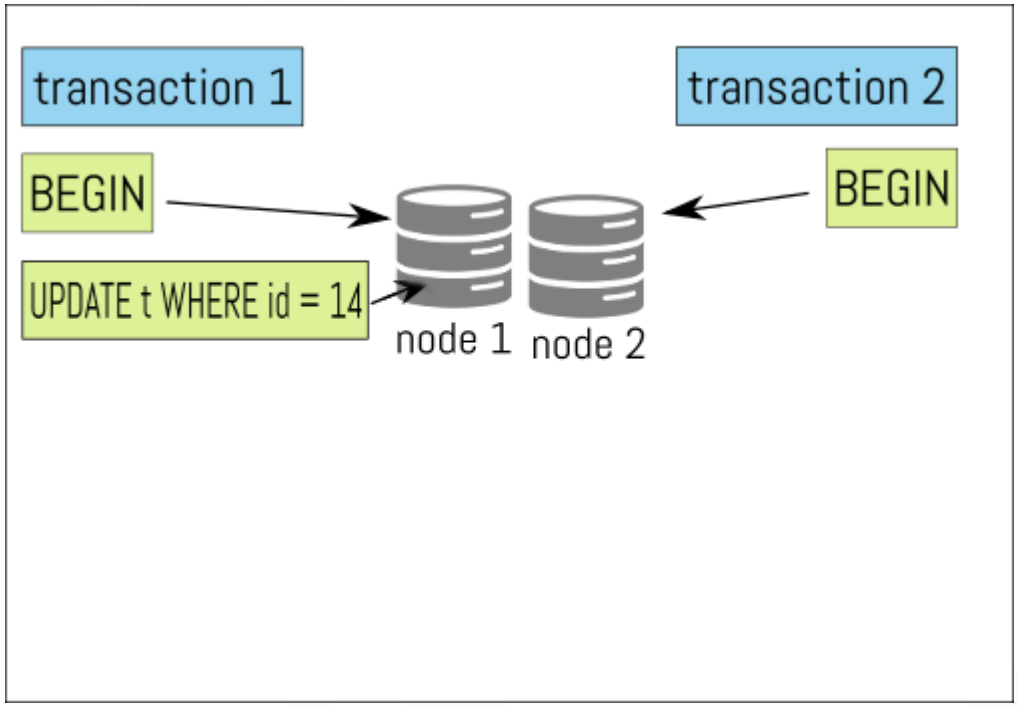
Traditional locking



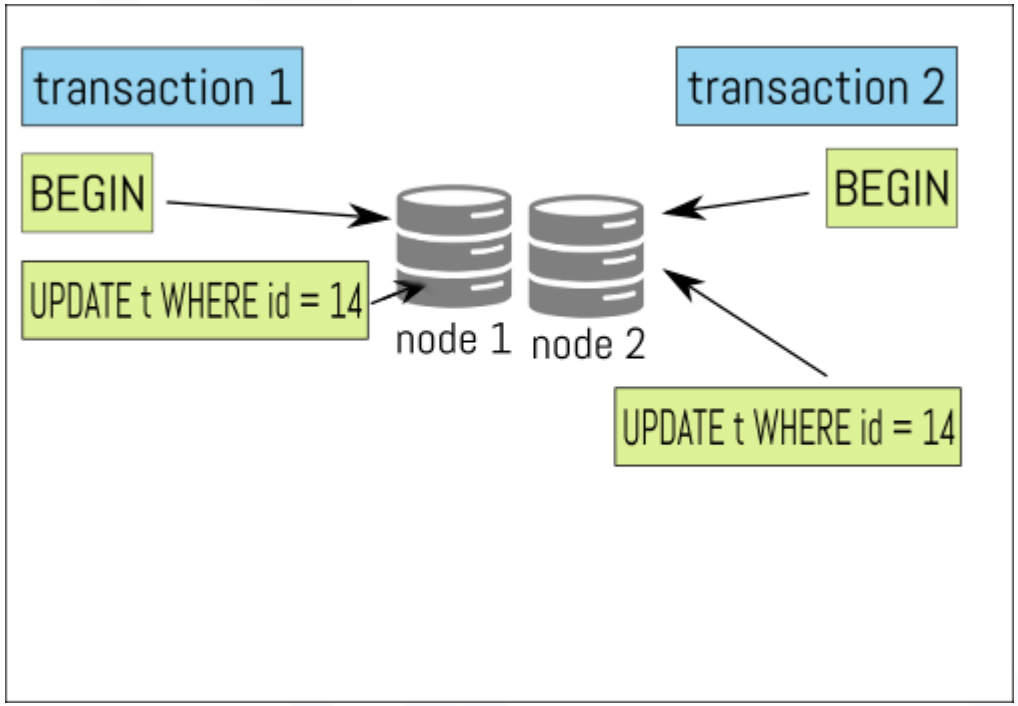
Optimistic Locking



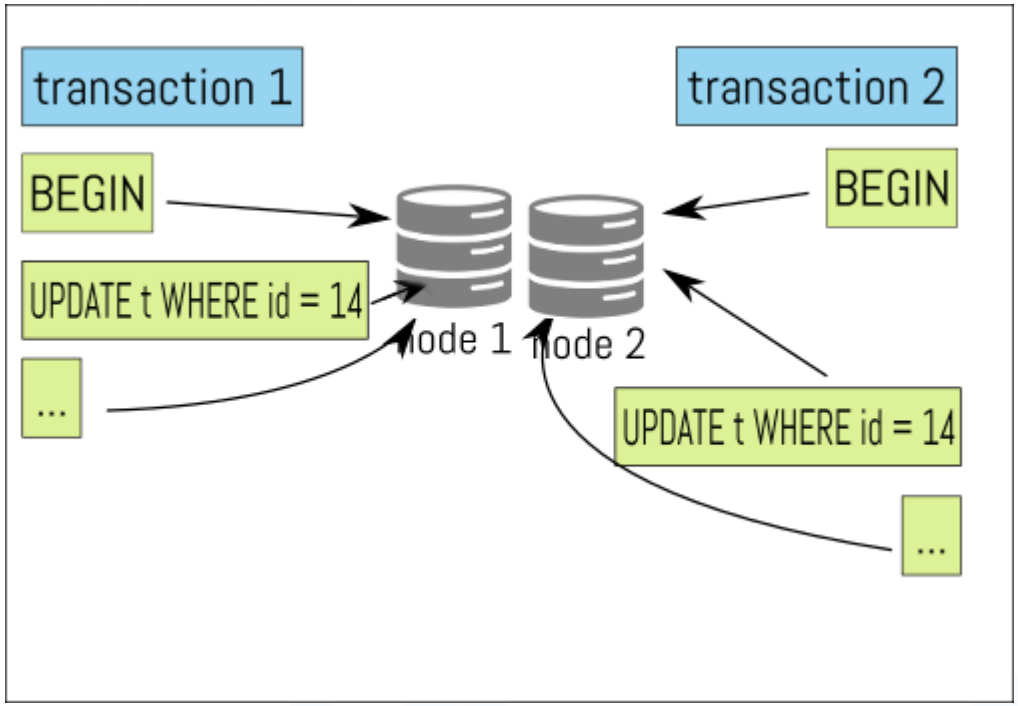
Optimistic Locking



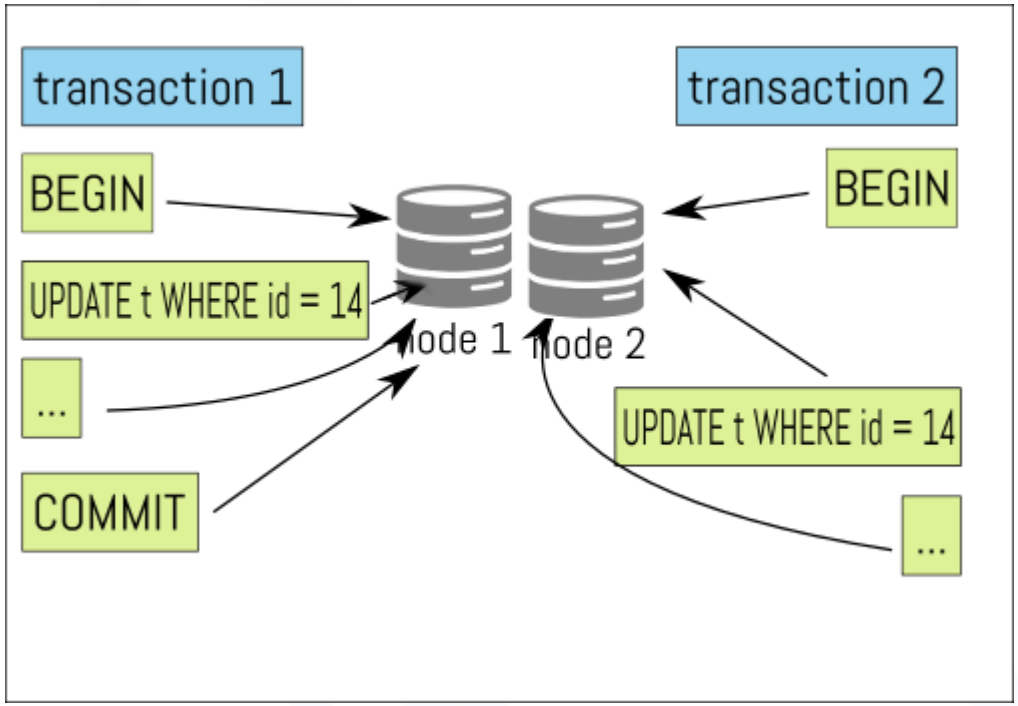
Optimistic Locking



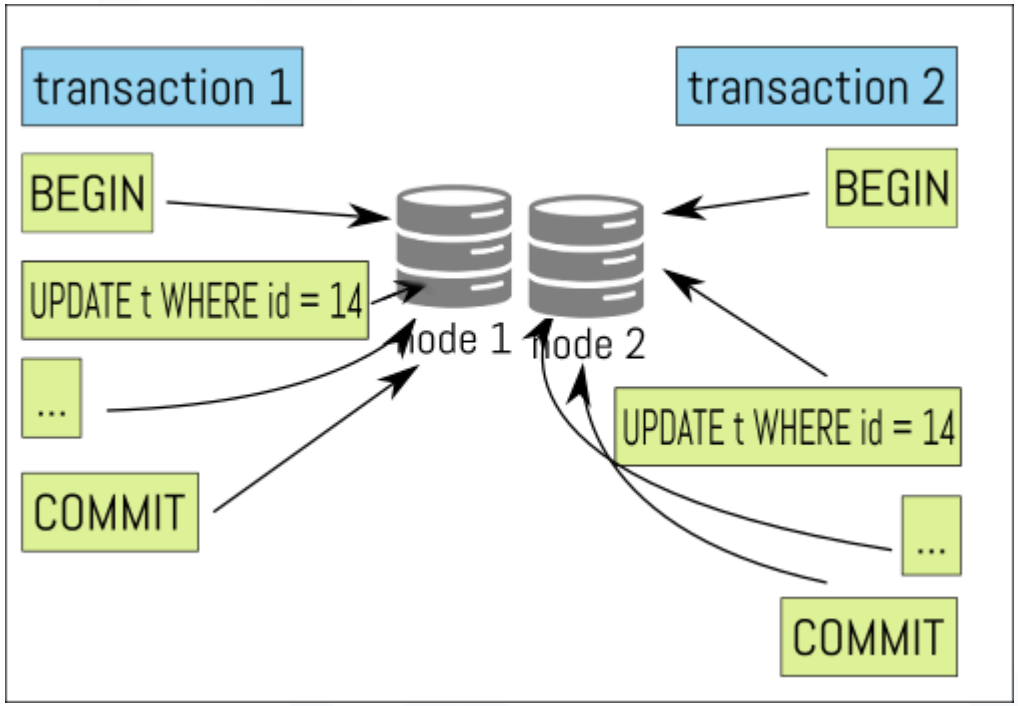
Optimistic Locking



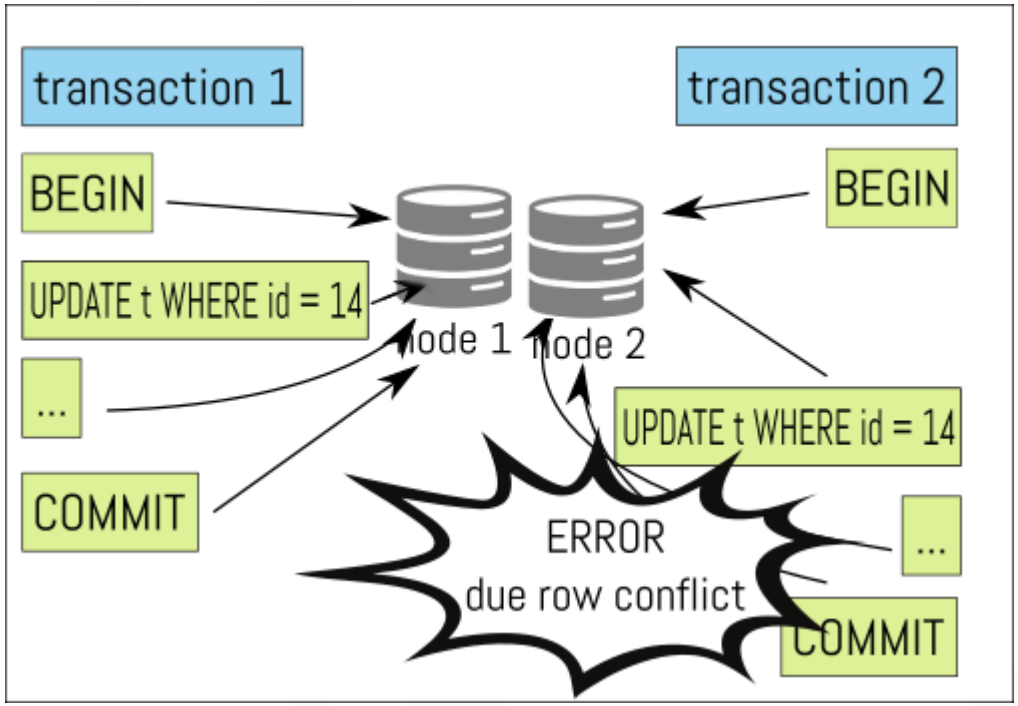
Optimistic Locking



Optimistic Locking



Optimistic Locking



The system returns error 149 as certification failed:

```
ERROR 1180 (HY000): Got error 149 during COMMIT
```

Such conflicts happen only when using multi-primary group !

not totally true in MySQL < 8.0.13 when failover happens

Drawbacks of optimistic locking

having a first-committer-wins system means conflicts will more likely happen when writing on multiple members with:

Drawbacks of optimistic locking

having a first-committer-wins system means conflicts will more likely happen when writing on multiple members with:

- large transactions

Drawbacks of optimistic locking

having a first-committer-wins system means conflicts will more likely happen when writing on multiple members with:

- large transactions
- long running transactions

Drawbacks of optimistic locking

having a first-committer-wins system means conflicts will more likely happen when writing on multiple members with:

- large transactions
- long running transactions
- hotspot records

can the transaction be committed ?

Certification

Certification

Certification is the process that only needs to answer the following unique question:

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic
- results are not reported to the group (does not require a new communication step)

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic
- results are not reported to the group (does not require a new communication step)
  - pass: commit/queue to appy

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic
- results are not reported to the group (does not require a new communication step)
  - pass: commit/queue to appy
  - fail: rollback/drop the transaction

Certification

Certification is the process that only needs to answer the following unique question:

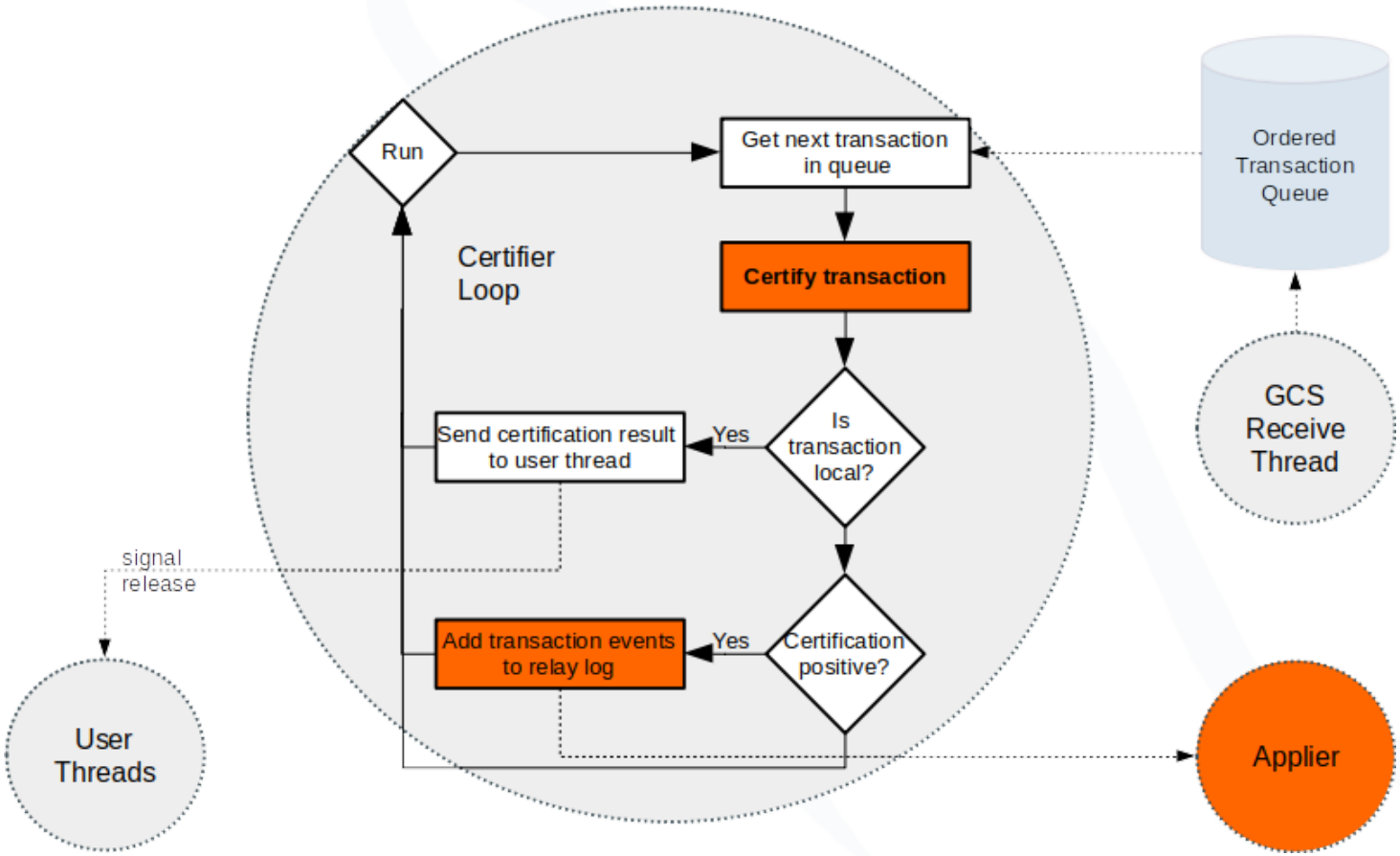
- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic
- results are not reported to the group (does not require a new communication step)
  - pass: commit/queue to appy
  - fail: rollback/drop the transaction
- serialized by the total order in GCS/XCOM + GTID

Certification

Certification is the process that only needs to answer the following unique question:

- *can the write (transaction) be committed ?*
  - based on yet to be applied transactions
  - such conflicts must come for other members/nodes
- happens on every member/node and is deterministic
- results are not reported to the group (does not require a new communication step)
  - pass: commit/queue to appy
  - fail: rollback/drop the transaction
- serialized by the total order in GCS/XCOM + GTID
- cost is based on trx size (# rows & # keys)

Certification



Houston we have a problem !

Flow Control

Flow Control

In Group Replication, every member send statistics about its queues (applier queue and certification queue) to the other members. Then every node decide to slow down or not if they realize that one node reached the threshold for one of the queue.

Flow Control

In Group Replication, every member send statistics about its queues (applier queue and certification queue) to the other members. Then every node decide to slow down or not if they realize that one node reached the threshold for one of the queue.

So when `group_replication_flow_control_mode` is set to *QUOTA* on the node seeing that one of the other members of the cluster is lagging behind (threshold reached), it will throttle the write operations to the a quota that is calculated based on the number of transactions applied in *the last second*, and then it is reduced below that by subtracting the "over the quota" messages from the last period.

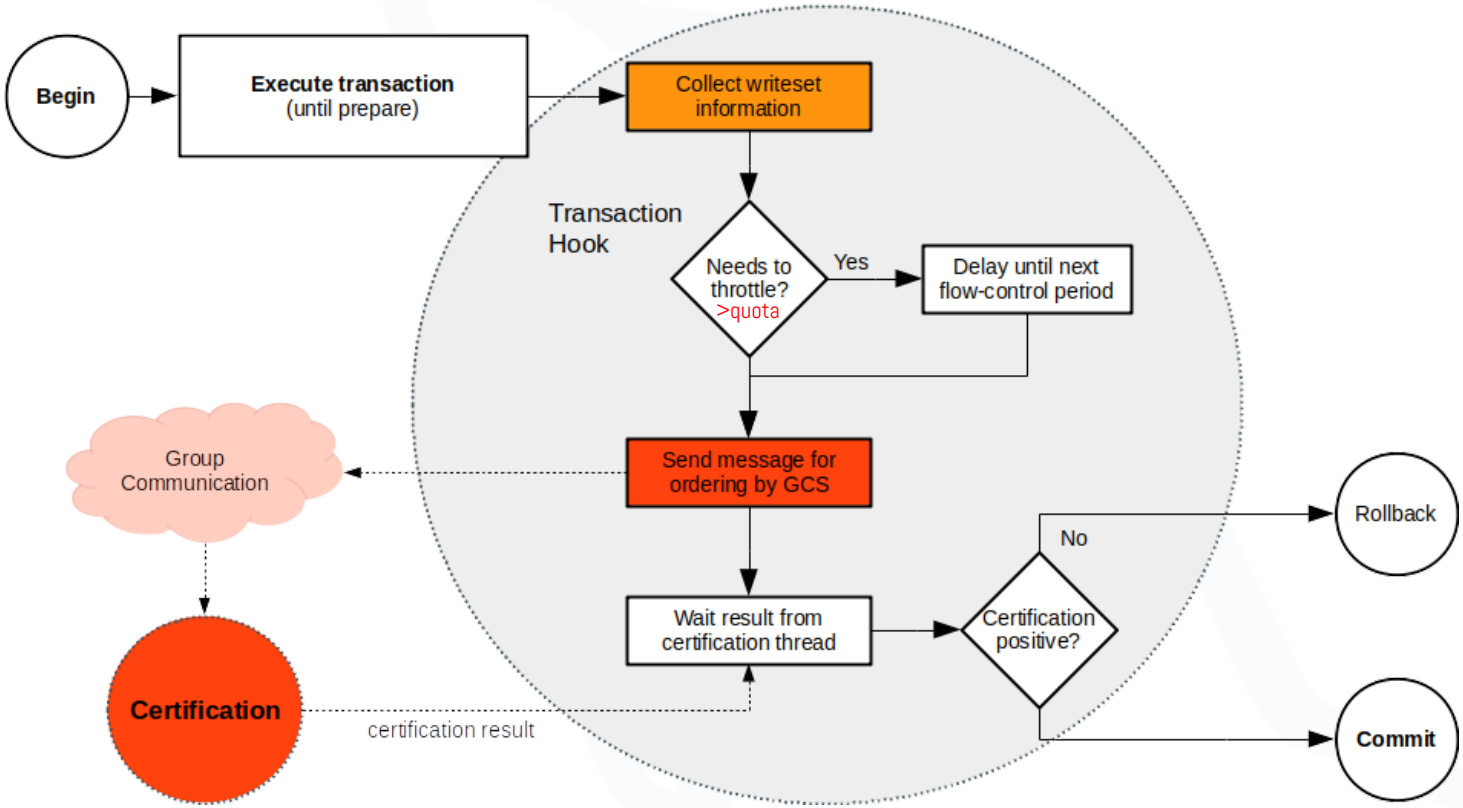
Flow Control

In Group Replication, every member send statistics about its queues (applier queue and certification queue) to the other members. Then every node decide to slow down or not if they realize that one node reached the threshold for one of the queue.

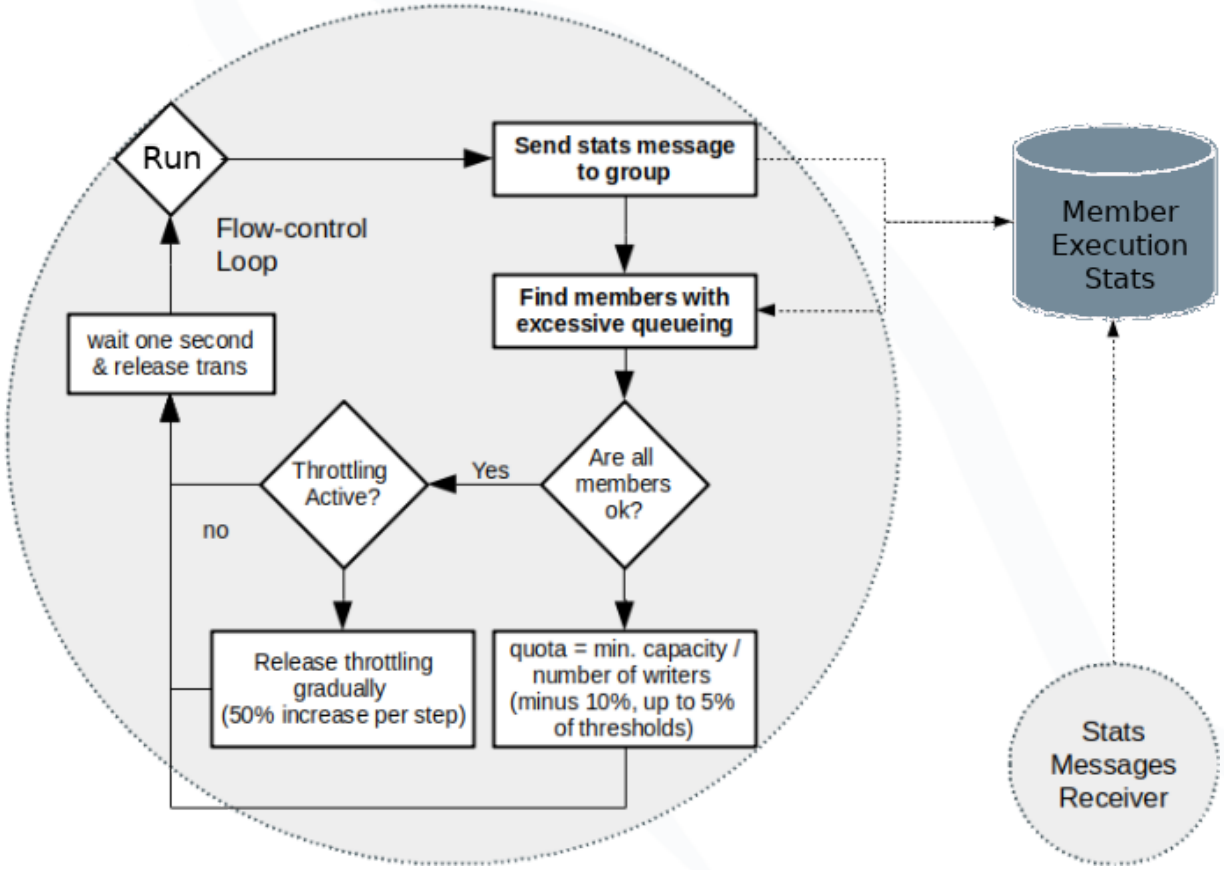
So when `group_replication_flow_control_mode` is set to *QUOTA* on the node seeing that one of the other members of the cluster is lagging behind (threshold reached), it will throttle the write operations to the a quota that is calculated based on the number of transactions applied in *the last second*, and then it is reduced below that by subtracting the "over the quota" messages from the last period.

This mean that the threshold is *NOT* decided on the node being slow, but the node writing a transaction checks its threshold flow control values and compare them to the statistics from the other nodes to decide to throttle or not.

Flow Control - on writer



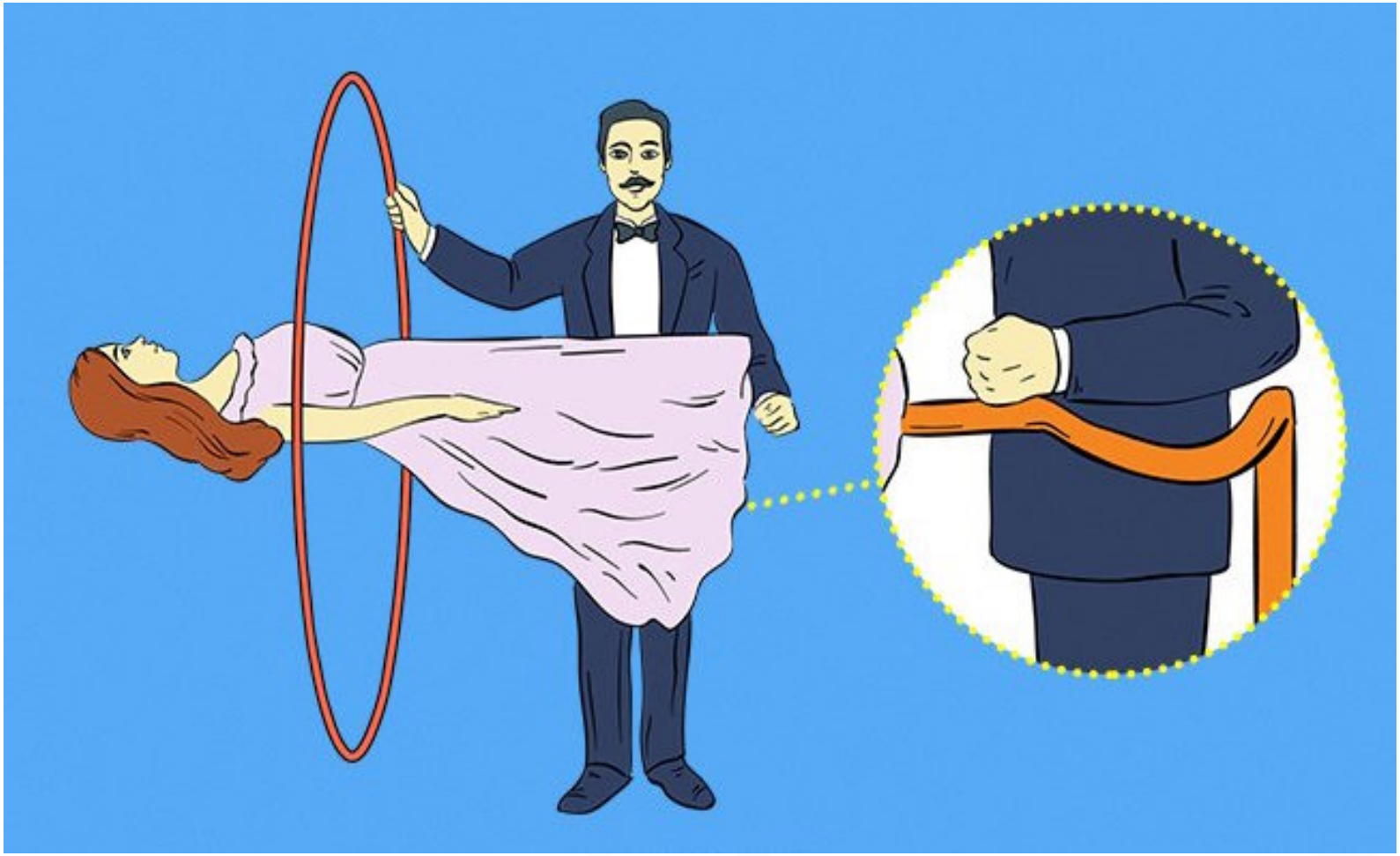
Flow Control - on all members



Flow Control - configuration variables

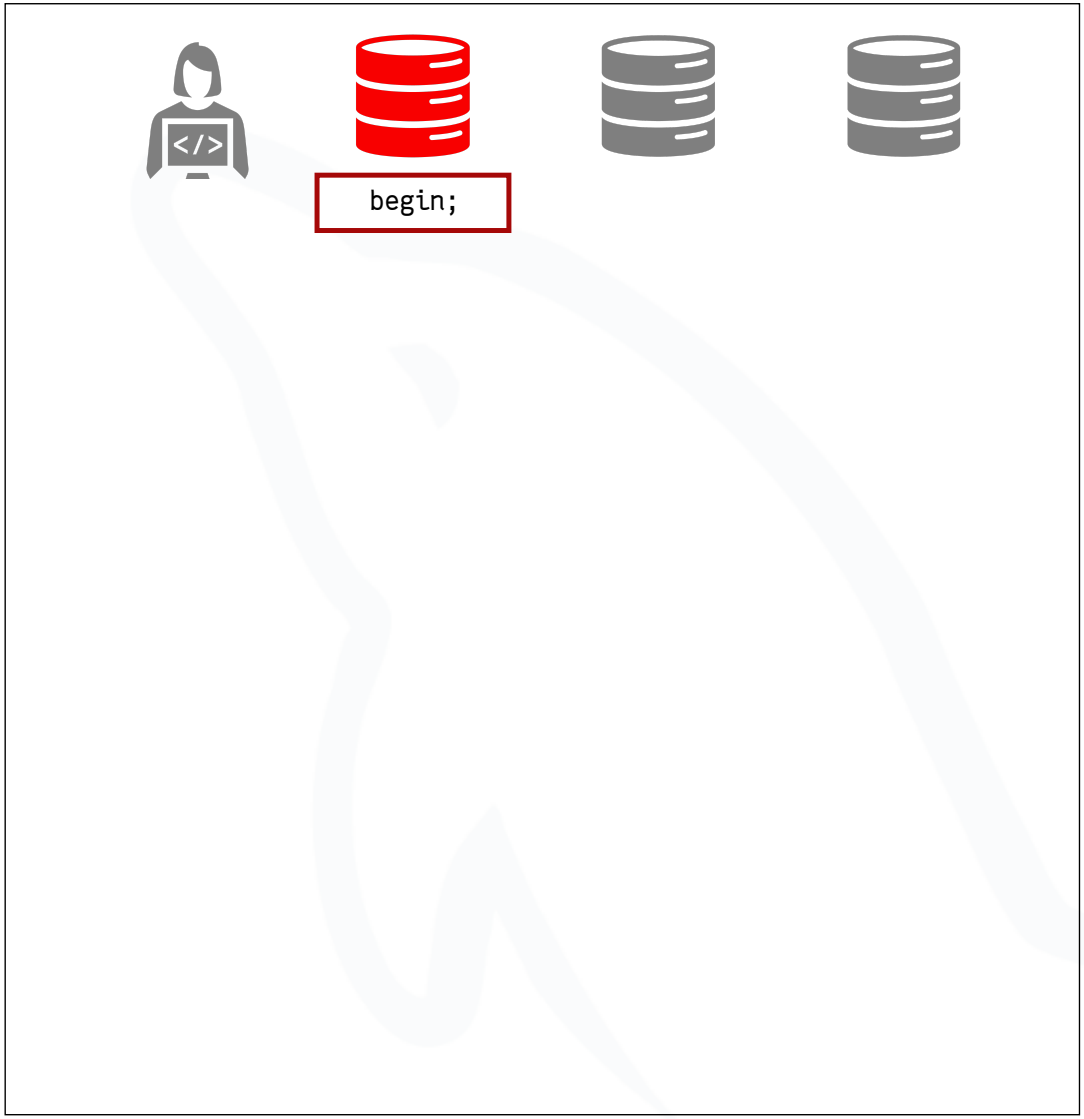
As in MySQL 8.0.13:

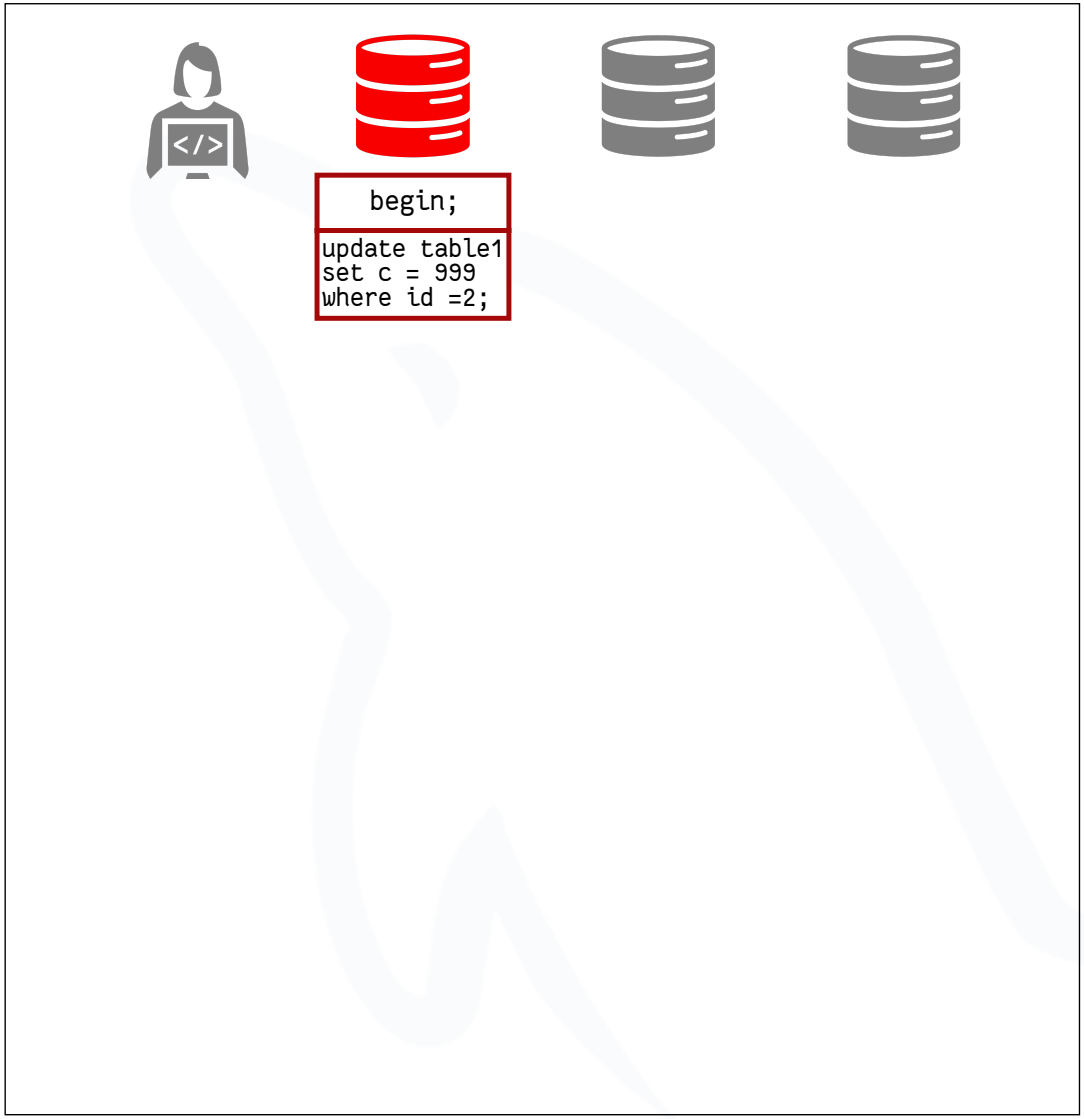
```
+-----------------------------------------------------+-------+
| Variable_name                                       | Value |
+-----------------------------------------------------+-------+
| group_replication_flow_control_applier_threshold    | 25000 |
| group_replication_flow_control_certifier_threshold  | 25000 |
| group_replication_flow_control_hold_percent         | 10    |
| group_replication_flow_control_max_quota            | 0     |
| group_replication_flow_control_member_quota_percent | 0     |
| group_replication_flow_control_min_quota            | 0     |
| group_replication_flow_control_min_recovery_quota   | 0     |
| group_replication_flow_control_mode                 | QUOTA |
| group_replication_flow_control_period               | 1     |
| group_replication_flow_control_release_percent      | 50    |
+-----------------------------------------------------+-------+
```

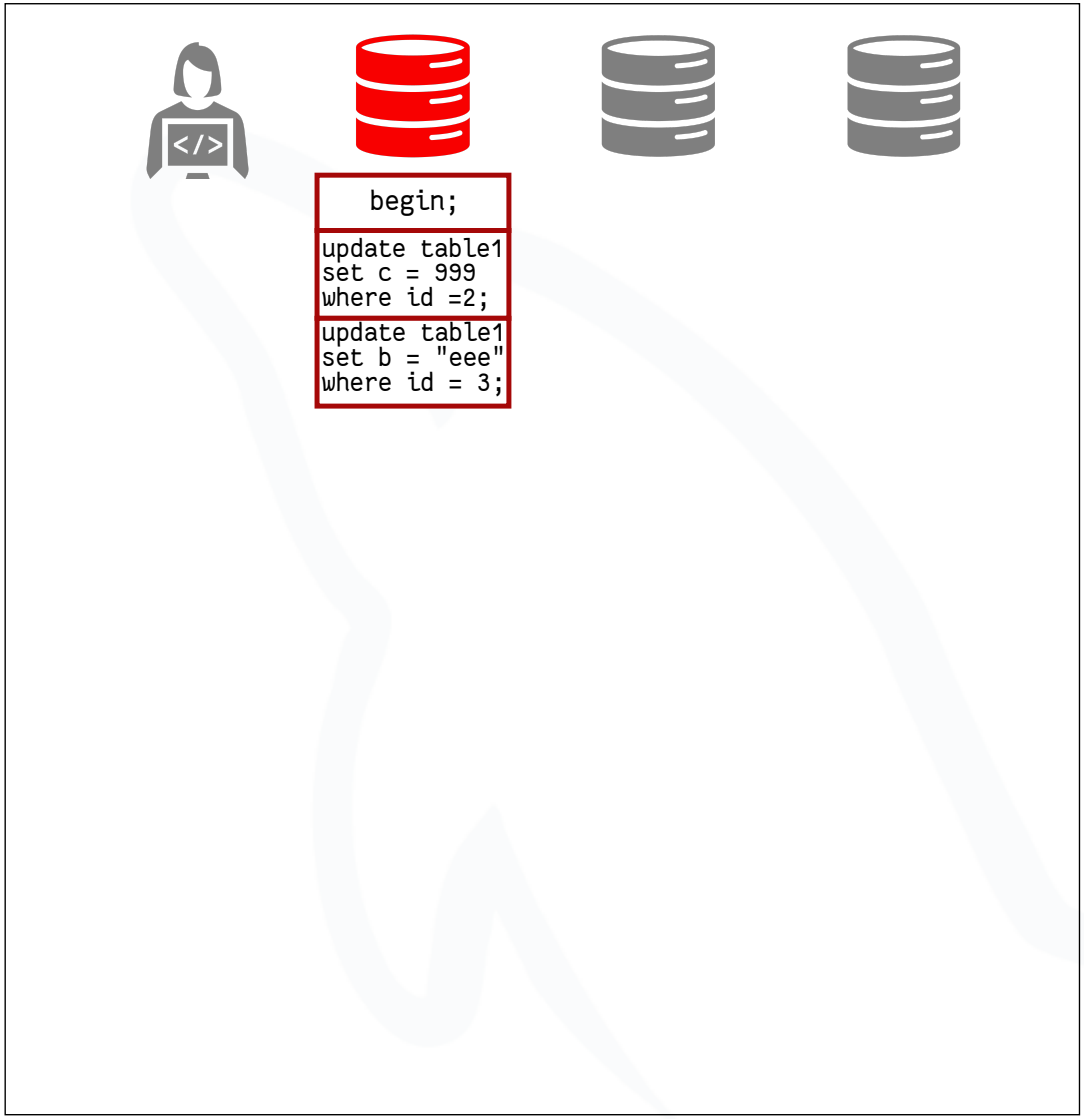


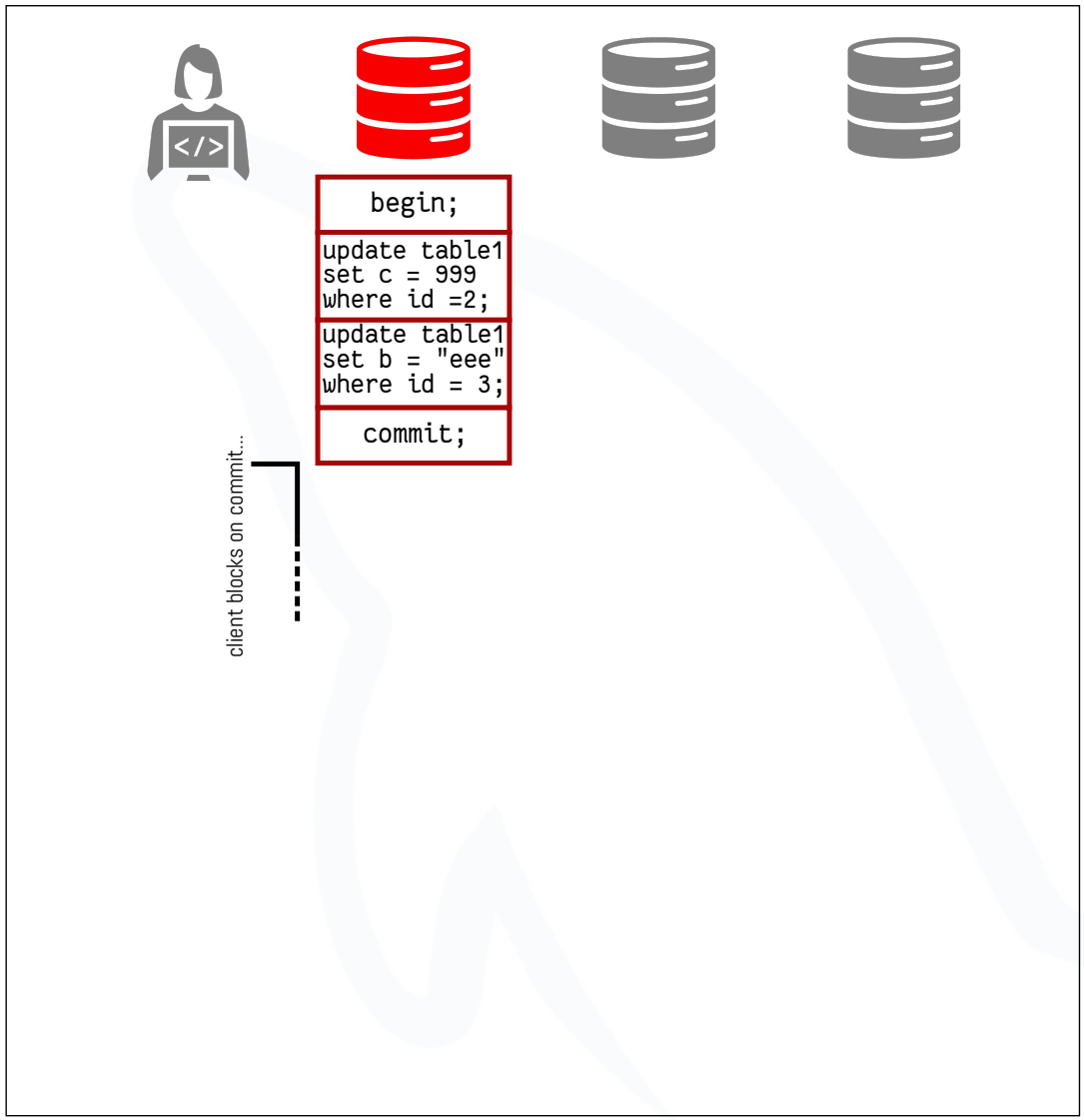
transaction's lifecycle in Group Replication

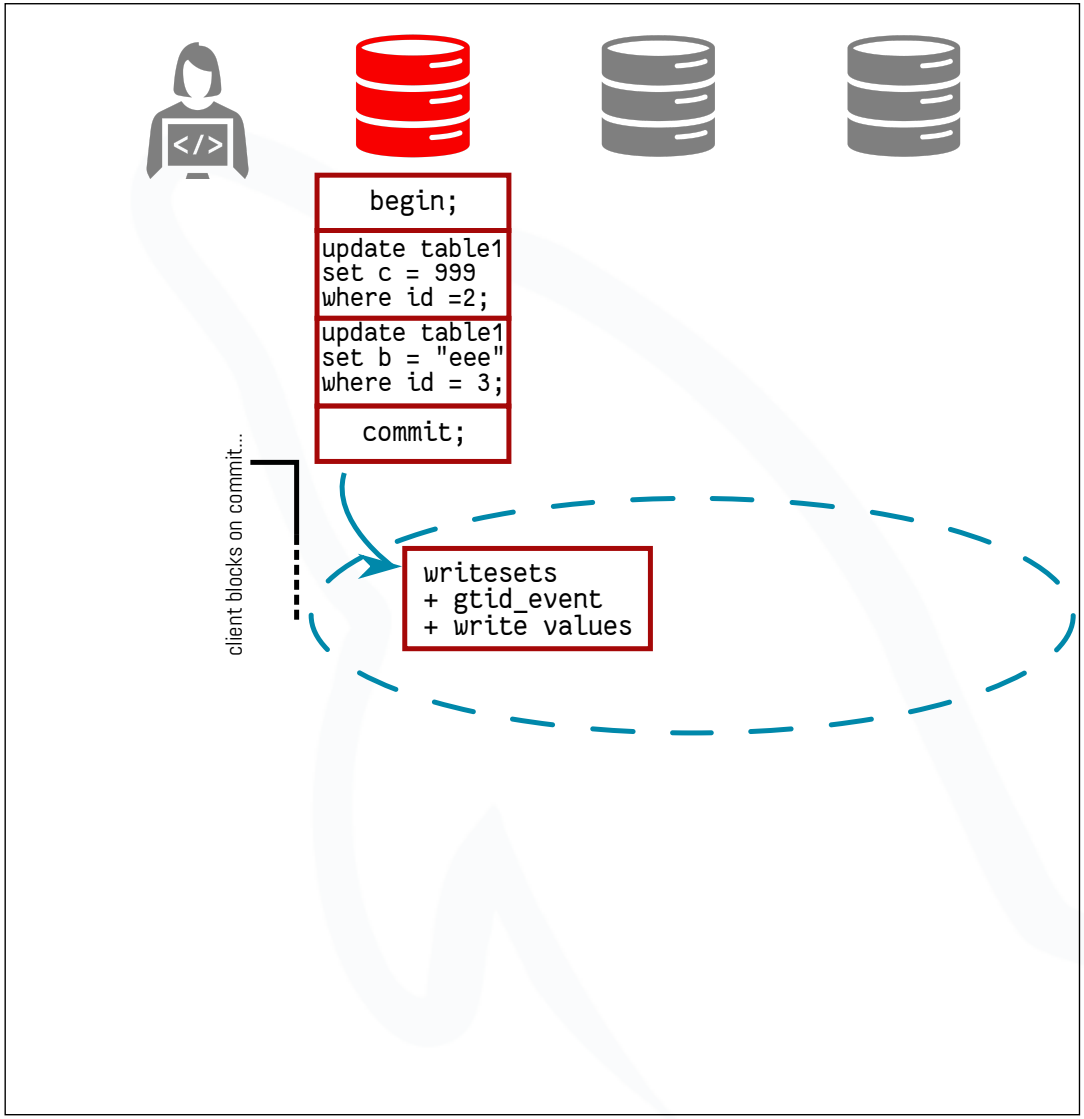
Summary

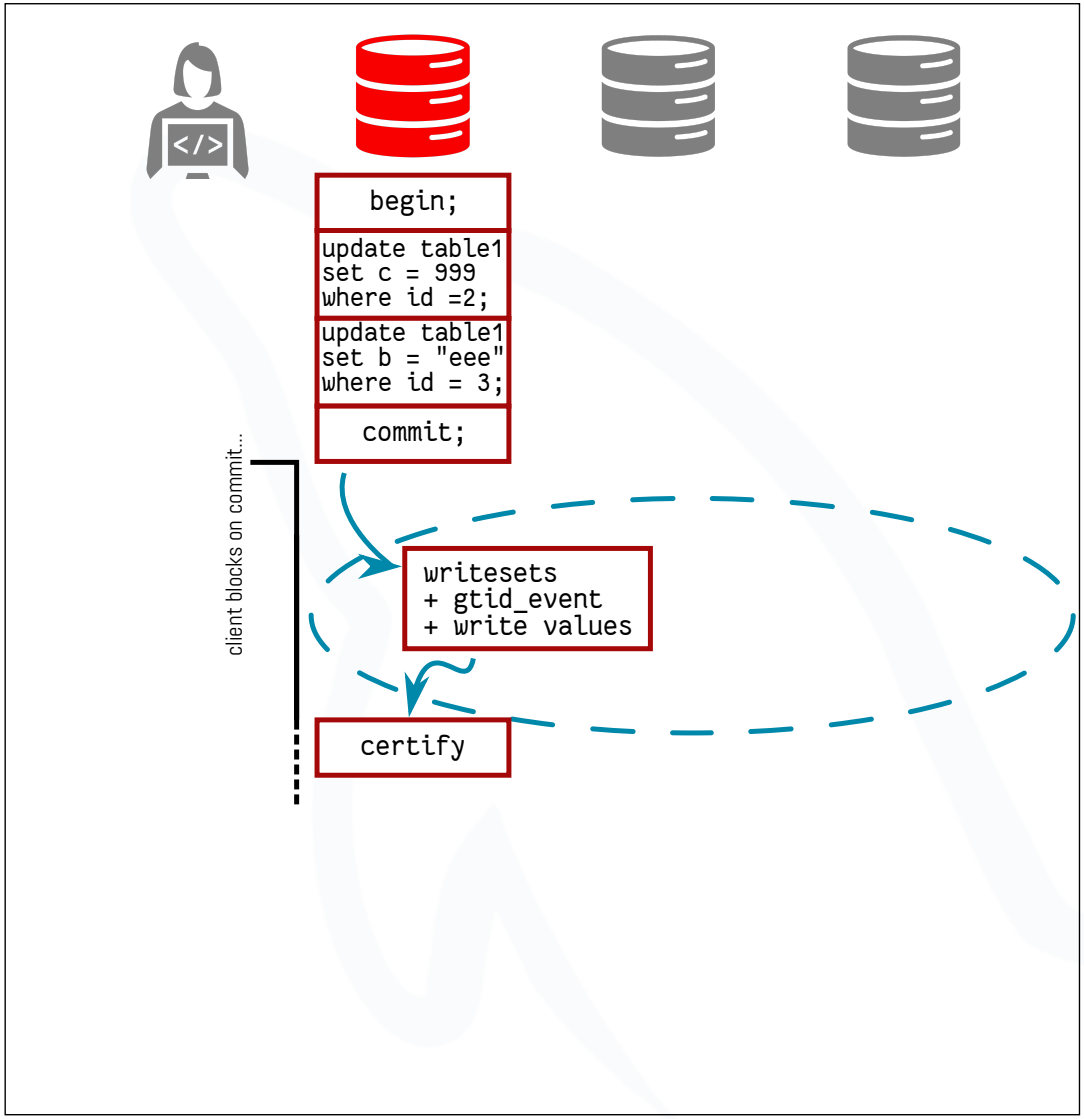


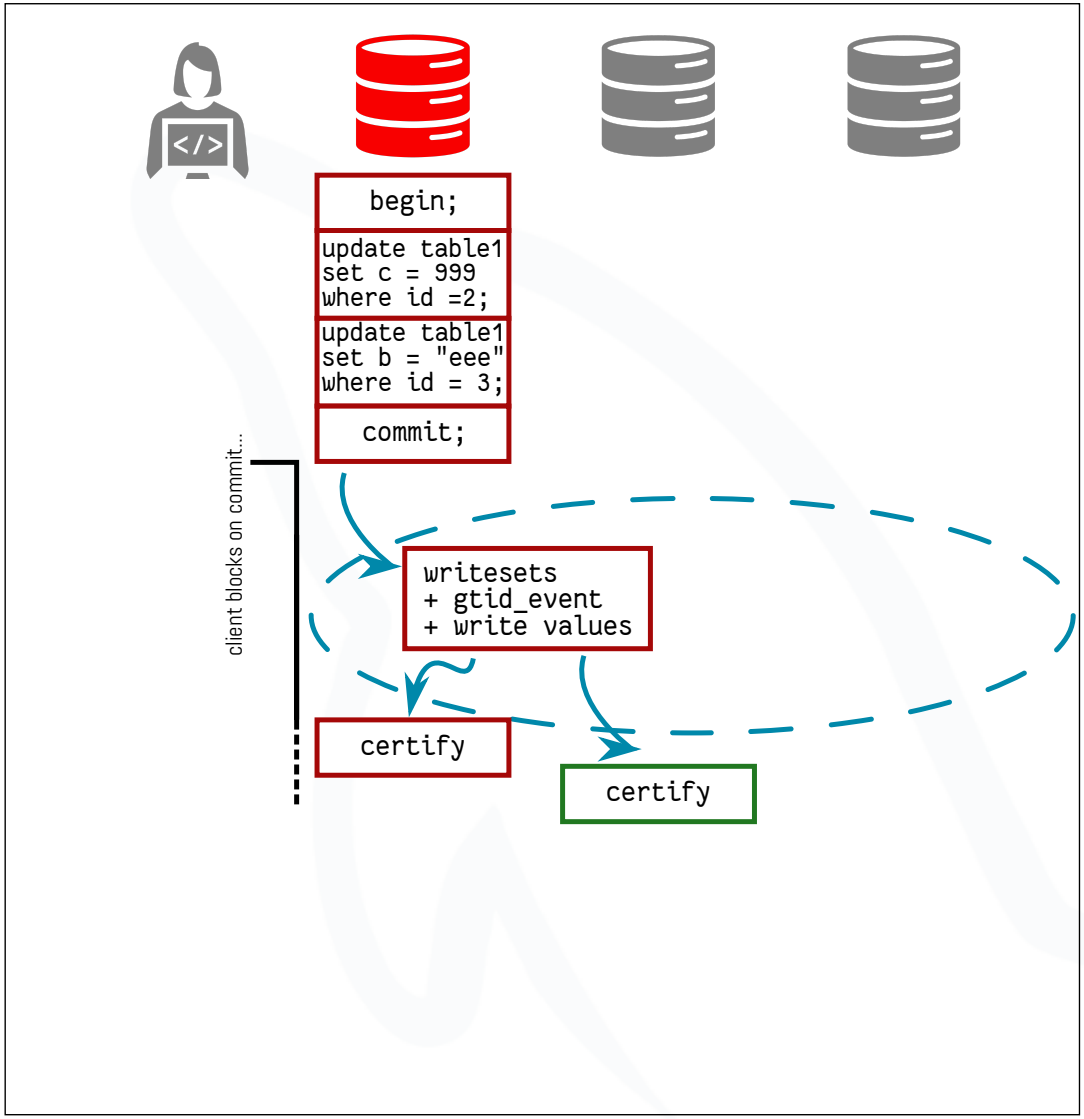


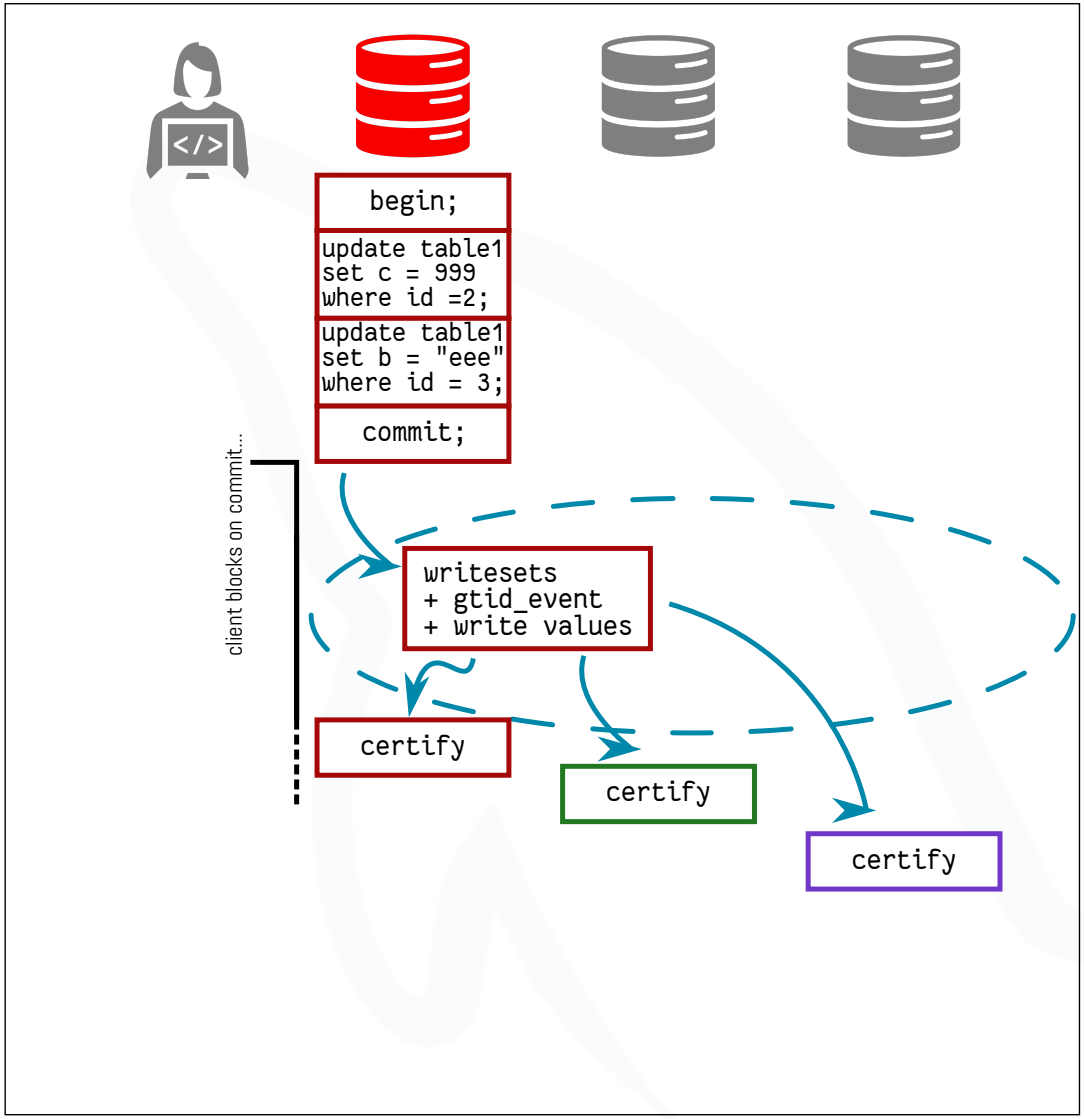


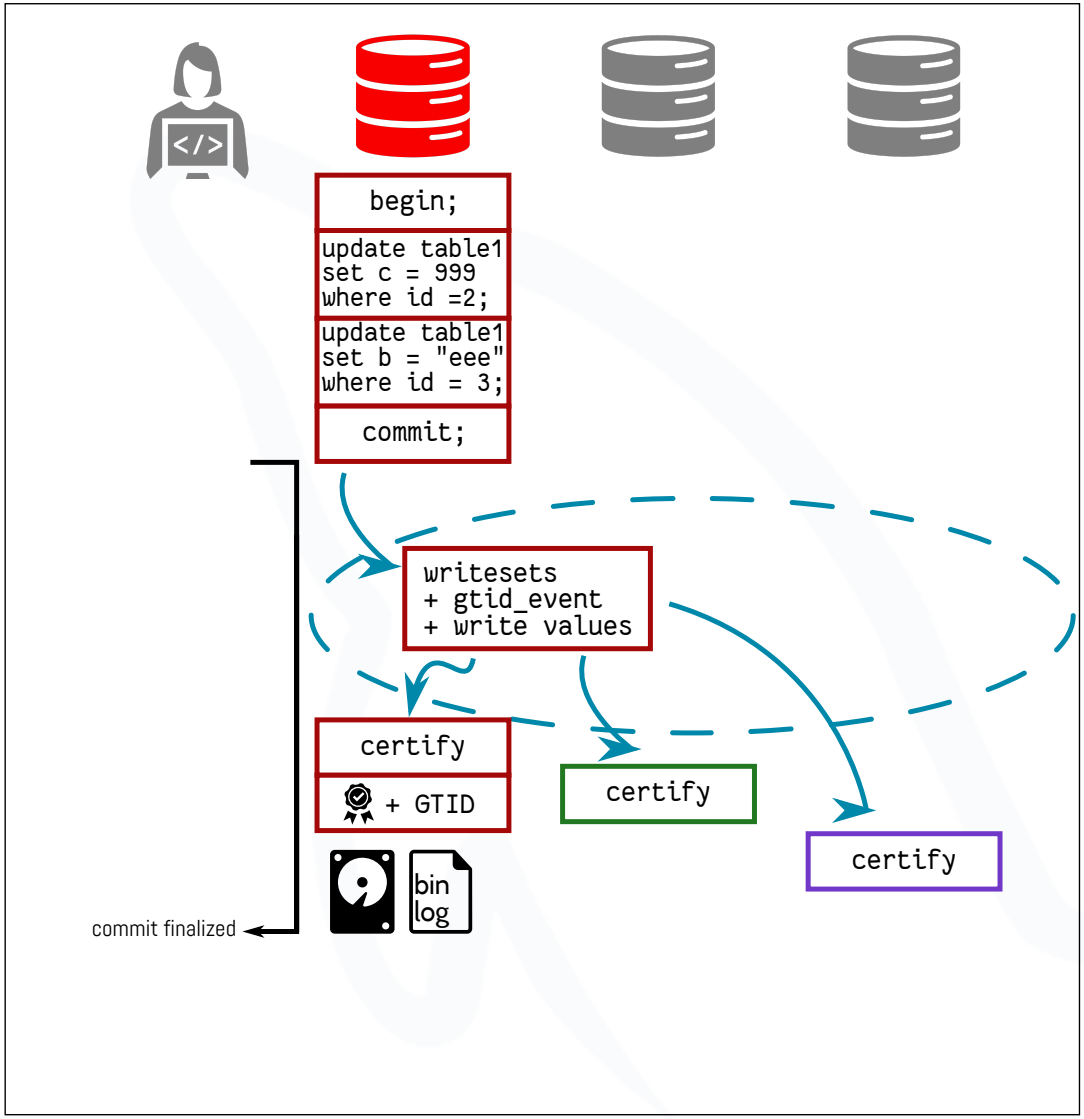


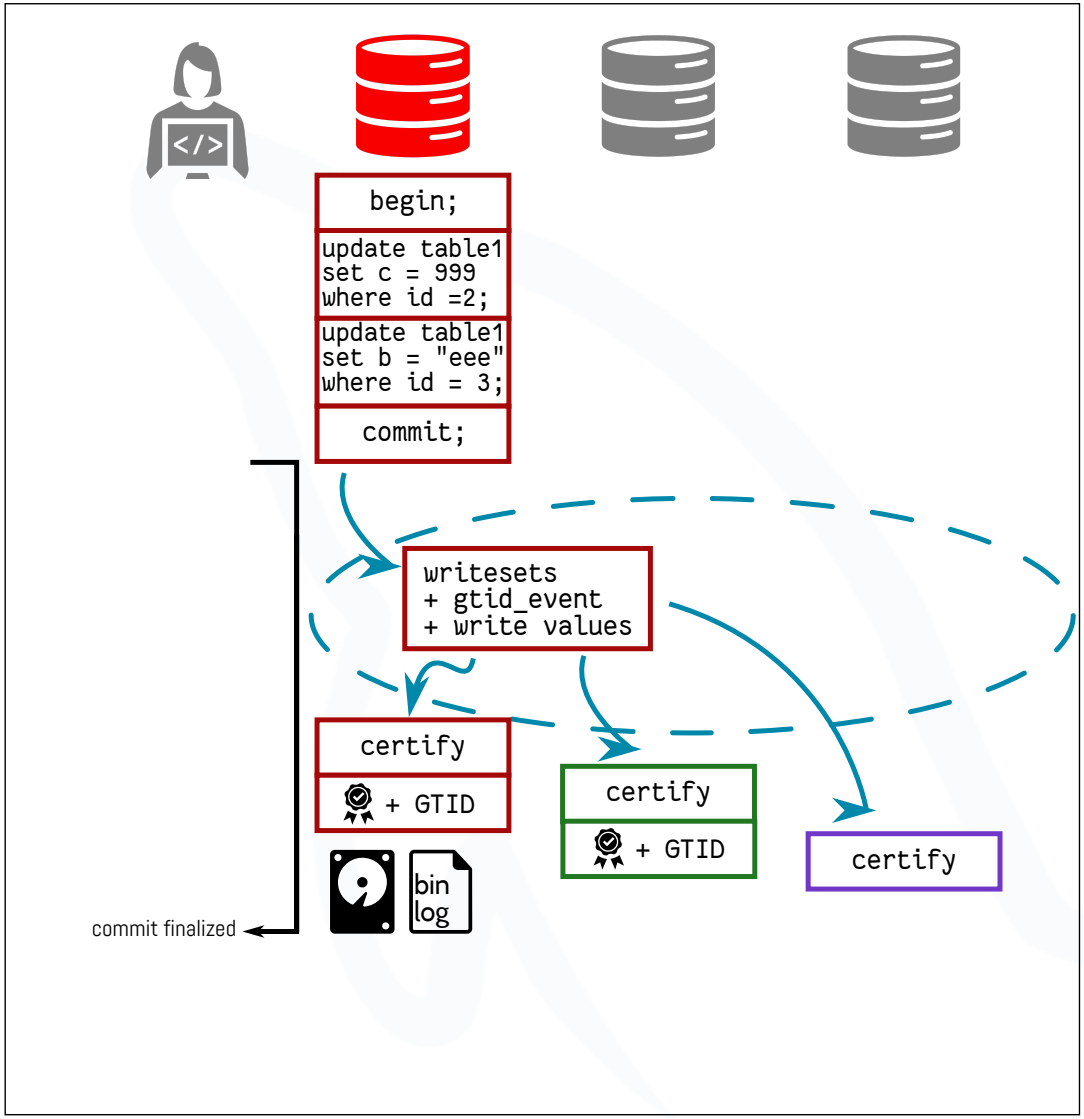


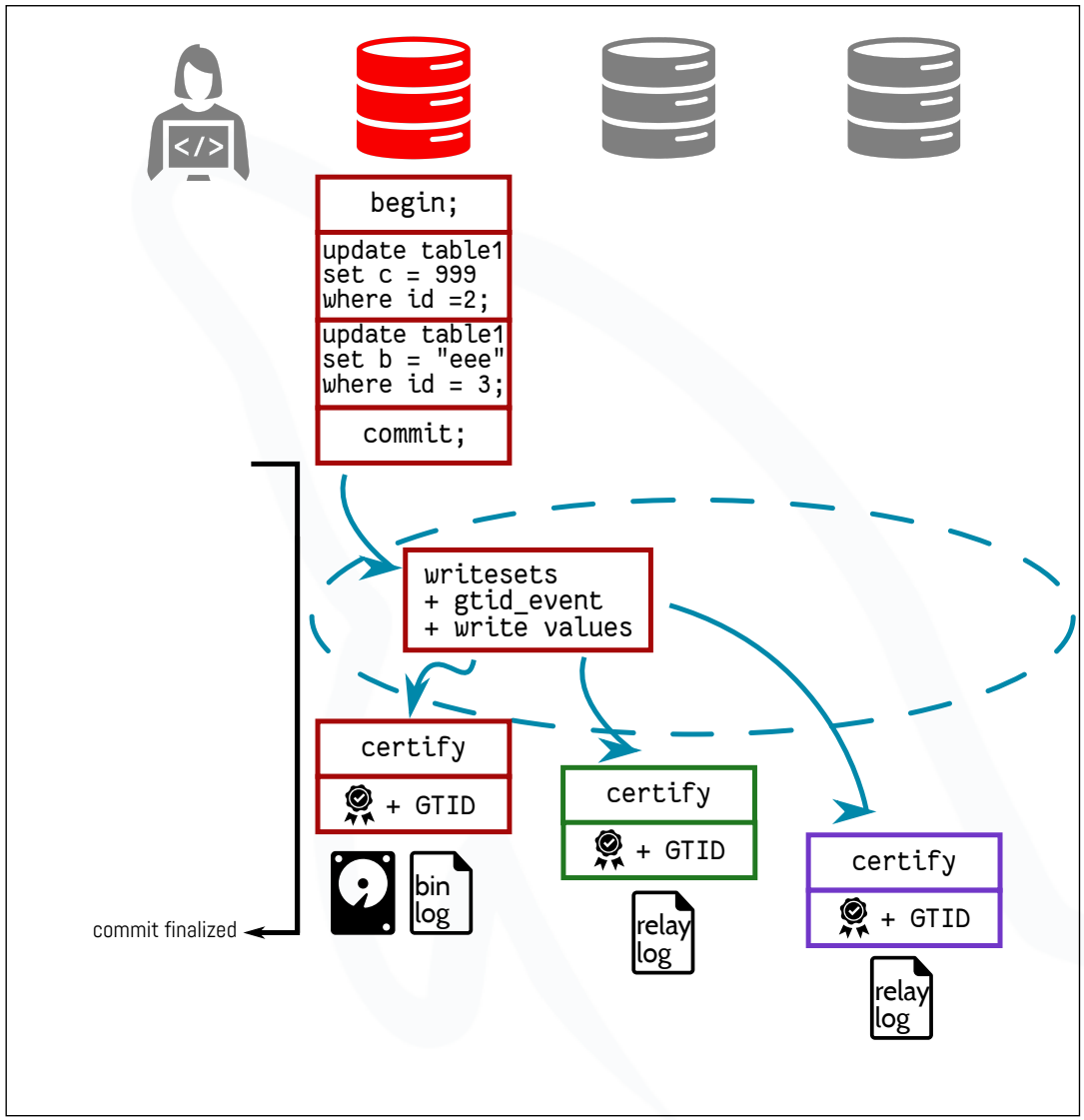


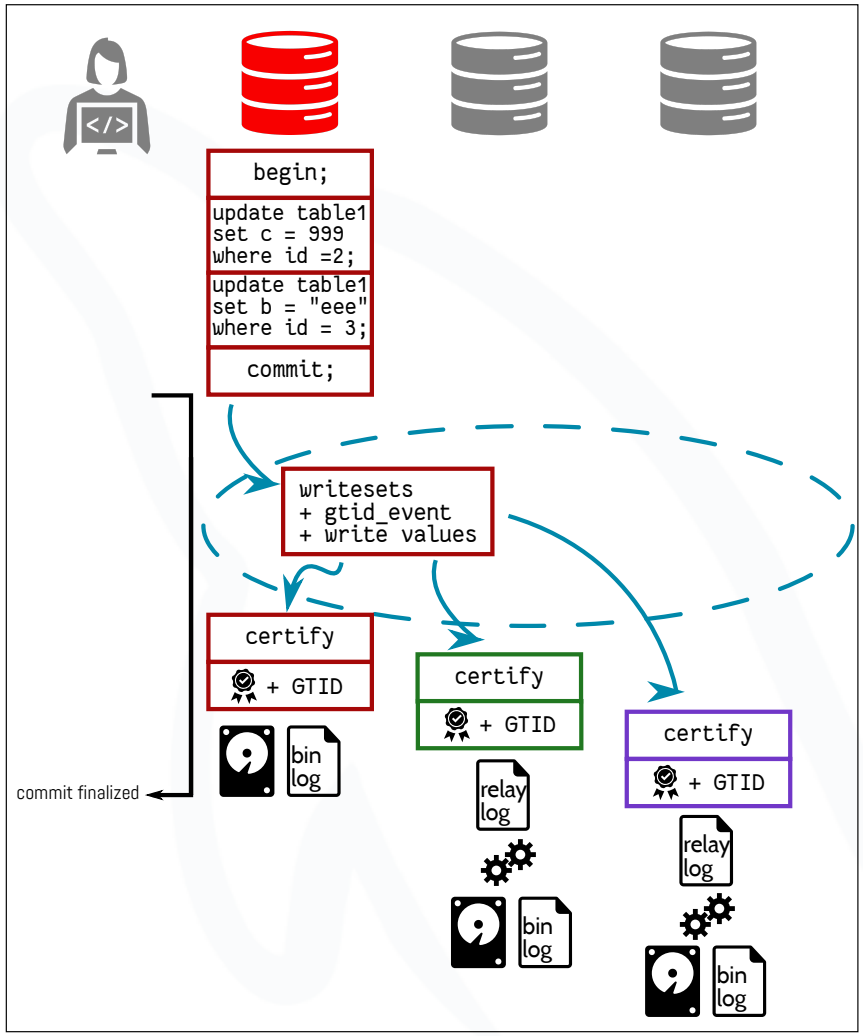












Thank you !

Any Questions ?