MySQL Group Replication
The Magic Explained v2

Frédéric Descamps
@lefred
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.
Who am I?

about.me/lefred
Frédéric Descamps

- @lefred
- MySQL Evangelist
- Managing MySQL since 3.23
- devops believer
- living in Belgium 🇧🇪
- https://lefred.be
Group Replication: heart of **MySQL** InnoDB Cluster

App Servers with **MySQL Router**

**MySQL Shell**
Setup, Manage, Orchestrate

**MySQL Group Replication**
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

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

```
table1
1  aaa  123
2  bbb  456
3  ccc  789
4  ddd  111
```
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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
# Writeset - examples

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
```

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td>UNI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name2</td>
<td>binary(1)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

*Copyright © 2018 Oracle and/or its affiliates. All rights reserved.*
Writeset - examples (2)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td>UNI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name2</td>
<td>binary(1)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td>UNI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name2</td>
<td>binary(1)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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

pke: PRIMARY | test | t3 | 1 | 1     | hash: 79134815725924853
pke: name    | test | t3 | 2 | 1     | hash: 11034644986657565027
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    |       |
+--------+-----------+------+-----+---------+-------+
```

```sql
mysql> insert into t3 values (1,2,3);
pke: PRIMARY | test |t3 | 1 | 1    hash: 79134815725924853
pke: name    | test |t3 | 2        hash: 11034644986657565827
```

```sql
mysql> update t3 set name=3 where id=1;
```
Writeset - examples (2)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>binary(1)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>binary(2)</td>
<td>YES</td>
<td>UNI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>name2</td>
<td>binary(1)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

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

```sql
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 | 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...

A kind of Magic
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 node 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...
IT'S NOT A TRICK. IT'S MAGIC.
MySQL Group Replication

Node 1
Node 2
Node 3

Statement (DML)
MySQL Group Replication

node 1

statement (DML)

node 2

node 3

transaction delivery
MySQL Group Replication

node 1

node 2

node 3

send

Z

O

M

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
MySQL Group Replication

node 1

node 2

node 3

send

ack

ZOOM
MySQL Group Replication

node 1  send  node 2

we can already go to next step

node 3
MySQL Group Replication

Node 1, Node 2, Node 3

Send

Ack

1 + 1 = majority

We can already go to next step
MySQL Group Replication

node 1

node 2

node 3

send

ack

ack

ack
MySQL Group Replication

node 1

statement (DML)

transaction delivery

certify
MySQL Group Replication

- node 1
- node 2
- node 3

statement (DML)

transaction delivery

certify
certify
MySQL Group Replication

Transaction delivery

- Statement (DML)
- Certify
- Commit finalized
- Begin
- Apply
- Certify
- Begin

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
MySQL Group Replication

transaction delivery

node 1

statement (DML)
certify
commit finalized

node 2
certify
begin
apply
commit finalized

certify
begin
apply
commit finalized

node 3
MySQL Group Replication (full transaction)
MySQL Group Replication (full transaction)

```
node 1  node 2  node 3
begin
statement
```
MySQL Group Replication (full transaction)

node 1
begin
statement
statement ...
node 2
node 3
MySQL Group Replication (full transaction)

node 1
begin
statement
statement ...
commit

node 2

node 3

client blocks on commit
MySQL Group Replication (full transaction)
MySQL Group Replication (full transaction)

node 1
begin
statement
statement..
commit

node 2

node 3

client blocks on commit
replicate

certify
certify

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
MySQL Group Replication (full transaction)
MySQL Group Replication (full transaction)

- Begin
- Statement
- Statement ...
- Commit

Client blocks on commit

-Replicate
- Certify
- Commit finalized

Node 1

Node 2

Node 3
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.
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

commit

message queue

time

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: Total Order Delivery - GTID

commit message

queue time
Group Replication: Total Order Delivery - GTID

commit

message queue

time

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: Total Order Delivery - GTID
Group Replication: Total Order Delivery - GTID

commit

message queue

time

commit
Group Replication: Total Order Delivery - GTID

commit

message queue

time

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: Total Order Delivery - GTID

commit

ack

ack

ack

message queue

time

ack = message delivery ack
Group Replication: Total Order Delivery - GTID

commit

message queue

majority reached

ack

ack

ack = message delivery ack

time

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: Total Order Delivery - GTID

commit

ack = message delivery ack
Group Replication: Total Order Delivery - GTID

commit | ack
--------|--------
mypg1   | ack
mypg2   | ack
mypg3   | ack
mypg4   | ack
mypg5   | ack

commit = message delivery ack
Group Replication: Total Order Delivery - GTID

commit = message delivery ack
Group Replication: Total Order Delivery - GTID

ack = message delivery ack
Group Replication: Total Order Delivery - GTID

commit → ack → ack

message queue

ack = message delivery ack
Group Replication: Total Order Delivery - GTID

certification queue

GTID: Group UID + seqno

certification
Group Replication: Total Order Delivery - GTID

certification queue

GTID: Group UID + seqno

certification

OK

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: Total Order Delivery - GTID
Group Replication: Total Order Delivery - GTID

certification queue

GTID: Group UID + seqno

certification

GTID: UUID: 2
Group Replication: Total Order Delivery - GTID

certification queue

certification

GTID: Group UID + seqno

GTID: UUID: 3
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-bef0-00002718d305:1-355

New write on another node:

Executed_Gtid_Set: 0b5c746d-d552-11e8-bef0-00002718d305:1-355,1000354
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:

Master

execute > binlog > commit

Slave 1

relay log > apply > binlog > commit

Slave 2

relay log > apply > binlog > commit
Group Replication: return from commit (2)

Semi-Sync Replication:

Master

Slave 1

Slave 2

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
Group Replication: return from commit (3)

Group Replication (*): eventual
Group Replication: return from commit (4)

Group Replication (*):

Master 1

Master 2

Master 3

(*): before
Group Replication: return from commit (5)

Group Replication (\(\ast\)):

\(\ast\): after

Master 1
- execute
- certify
- binlog
- commit

Master 2
- certify
- relay log
- apply
- binlog
- commit

Master 3
- certify
- relay log
- apply
- binlog
- commit

Consensus

Copyright \(\text{\textcopyright} \ 2018\) Oracle and/or its affiliates. All rights reserved.
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()`

```sql
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

group replication write concurrency
Event Horizon

GCS Write Consensus Concurrency

group replication write 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
Magician Of The Year
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

transaction 1
BEGIN

single node
Traditional locking

```
BEGIN
UPDATE t WHERE id = 14
```

single node
Traditional locking

transaction 1
BEGIN
UPDATE t WHERE id = 14

transaction 2
BEGIN

single node
Traditional locking

```sql
transaction 1
BEGIN
UPDATE t WHERE id = 14
... single node

transaction 2
BEGIN
UPDATE t WHERE id = 14
```
Traditional locking

```
begin
update t where id = 14
...
```

```
begin
update t where id = 14
```

single node

`waits on COMMIT in trx 1`
Traditional locking

```
transaction 1
BEGIN
UPDATE t WHERE id = 14
...  
COMMIT

transaction 2
BEGIN
UPDATE t WHERE id = 14

single node
```
Optimistic Locking

transaction 1

BEGIN

node 1 node 2
Optimistic Locking

```
transaction 1
BEGIN
UPDATE t WHERE id = 14
```

```
transaction 2
BEGIN
```
Optimistic Locking

transaction 1
BEGIN
UPDATE t WHERE id = 14

node 1 node 2

BEGIN

transaction 2
UPDATE t WHERE id = 14
Optimistic Locking

```
transaction 1
BEGIN
UPDATE t WHERE id = 14
...
```

```
transaction 2
BEGIN
UPDATE t WHERE id = 14
...
```
Optimistic Locking

- Transaction 1
  - BEGIN
  - UPDATE t WHERE id = 14
  - ...
  - COMMIT

- Transaction 2
  - BEGIN
  - UPDATE t WHERE id = 14
  - ...

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
Optimistic Locking

transaction 1
BEGIN
UPDATE t WHERE id = 14
...
COMMIT

transaction 2
BEGIN
UPDATE t WHERE id = 14
...
COMMIT
Optimistic Locking

The system returns error 149 as certification failed:

```
ERROR 1180 (HY000): Got error 149 during COMMIT
```
Optimistic Locking (2)

The conflict detection can happen at two different stages:

- Only one of the two transactions was sent to the Group and the other one is still running (local).
- If both transactions were sent to the Group at almost the same time and both reached GCS/XCOM.
Optimistic Locking (3)

Only one transaction was sent to GCS and the other conflicting one is still local:

In this case, it’s the high priority transaction mechanism of InnoDB that *kills* the local one:

- If you try any statement in the transaction’s session you will see:

```
ERROR: 1213: Deadlock found when trying to get lock; try restarting transaction
```

- If you try to *commit* the transaction you will see:

```
ERROR: 1180: Got error 149 - 'Lock deadlock; Retry transaction' during COMMIT
```
Optimistic Locking (4)

Both transactions were sent to GCS:

The second transaction (the conflicting one) will return:

ERROR 3101 (40000): Plugin instructed the server to rollback the current transaction.
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
Configurable Consistency Guarantees

Consistency Levels
Consistency: EVENTUAL (default)

By default, there is no synchronization point for the transactions, when you perform a write on a node, if you immediately read the same data on another node, it is eventually there.

```
mysql> show variables like 'group_replication_consistency';
+--------------------------+----------+
| Variable_name            | Value    |
+--------------------------+----------+
| group_replication_consistency | EVENTUAL |
+--------------------------+----------+
```

Consistency: EVENTUAL (default)

By default, there is no synchronization point for the transactions, when you perform a write on a node, if you immediately read the same data on another node, it is eventually there.

```
mysql> show variables like 'group_replication_consistency';
+--------------------------+----------+
| Variable_name            | Value    |
+--------------------------+----------+
| group_replication_consistency | EVENTUAL |
+--------------------------+----------+
```
Consistency: EVENTUAL (default)

By default, there is no synchronization point for the transactions, when you perform a write on a node, if you immediately read the same data on another node, it is eventually there.

```sql
mysql> show variables like 'group_replication_consistency';
+-----------------------------+----------+
| Variable_name              | Value    |
+-----------------------------+----------+
| group_replication_consistency | EVENTUAL |
+-----------------------------+----------+
```

Since MySQL 8.0.16, we have to possibility to set the synchronization point at read or at write or both (globally or for a session).
Consistency: BEFORE (READ)

T1: EVENTUAL

M1

execute

commit

M2

commit

commit

M3

commit

commit

commit

T2 will wait for T1 commit

T2: BEFORE
Consistency: AFTER (WRITE)

T1: AFTER

Commit is only done once all ONLINE members ack the prepare

T2 will wait for T1 commit

T2: EVENTUAL
Consistency: BEFORE_AND_AFTER

T2: BEFORE_AND_AFTER

T2 will wait for T1 commit

T2 commit is only done once all ONLINE members ack the prepare

M1:

commit

execute/prepare

commit

prepare

commit

execute/prepare

commit

execute/prepare

commit

M2:

commit

prepare

commit

M3:

execute

commit

T1: EVENTUAL

T3: EVENTUAL

T3 will wait for T2 commit
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 apply
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 apply
  - 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 apply
  - 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 apply
  - fail: rollback/drop the transaction
- serialized by the total order in GCS/XCOM + GTID
- cost is based on trx size (\# rows & \# keys)
Certification

- Run
  - Get next transaction in queue
  - Certifier Loop
    - Send certification result to user thread
    - Is transaction local?
      - Yes: Certification positive?
      - No: Add transaction events to relay log
      - Certification positive?
    - Certify transaction
  - Ordered Transaction Queue
  - GCS Receive Thread
  - Applier
  - User Threads
  - Signal release
Enhanced support for large transactions

Message Fragmentation
(1). If the message size exceeds the maximum size that the user allows (\texttt{group\_replication\_communication\_max\_message\_size}), the member fragments the message into chunks that do not exceed the maximum size.

(2). The member broadcasts each chunk to the group, i.e. forwards each chunk individually to XCom.
Reassembling an incoming message - *first fragment*

(2). The members conclude that the incoming message is actually a fragment of a bigger message.

(3). The members buffer the incoming fragment because they conclude the fragment is a chunk of a still-incomplete message. (Fragments contain the necessary metadata to reach this conclusion.)
Reassembling an incoming message - second fragment

(5). Same as step 3.

(6). Same as step 4.
Reassembling an incoming message - **last fragment**

1. The members conclude that the incoming message is actually a fragment of a bigger message.

2. The members conclude that the incoming fragment is the last chunk missing, reassemble the original, whole message, and process it.
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

1. Begin
2. Execute transaction (until prepare)
3. Collect writeset information
4. Transaction Hook
   - Needs to throttle? (quota)
   - Yes: Delay until next flow-control period
   - No: Send message for ordering by GCS
5. Certification
6. Wait result from certification thread
7. Certification positive?
   - Yes: Commit
   - No: Rollback
Flow Control - on all members

- Flow-control Loop
  - wait one second & release trans
  - Throttling Active?
    - no: Release throttling gradually (50% increase per step)
    - yes: Find members with excessive queueing
      - Are all members ok?
        - yes: Send stats message to group
        - no: flow-control Loop

- Member Execution Stats

- Stats Messages Receiver
Flow Control - configuration variables

As in MySQL 8.0.16:

<table>
<thead>
<tr>
<th>Variable_name</th>
<th>Value</th>
</tr>
</thead>
</table>
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    |
Summary

transaction's lifecycle in Group Replication
begin;
begin;
update table1
set c = 999
where id = 2;
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

client blocks on commit...
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

writesets
+ gtid_event
+ write values

client blocks on commit...
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

client blocks on commit...
writesets
+ gtid_event
+ write values
certify
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

writesets
+ gtid_event
+ write values

client blocks on commit...

certify

certify
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

writesets
+ gtid_event
+ write values

certify

certify

certify

certify
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
begin;
update table1 set c = 999 where id = 2;
update table1 set b = "eee" where id = 3;
commit;
writesets + gtid_event + write values
commit finalized
bin log
+ GTID
+ GTID
+ GTID
certify
certify
certify
Copyright © 2018 Oracle and/or its affiliates. All rights reserved.
begin;
update table1 set c = 999 where id = 2;
update table1 set b = "eee" where id = 3;
commit;

writesets  
+ gtid_event  
+ write values

certify  
+ GTID

commit finalized

certify  
+ GTID

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
begin;
update table1
set c = 999
where id = 2;
update table1
set b = "eee"
where id = 3;
commit;

Copyright @ 2018 Oracle and/or its affiliates. All rights reserved.
Thank you!

Any Questions?