

Innovation Everywhere

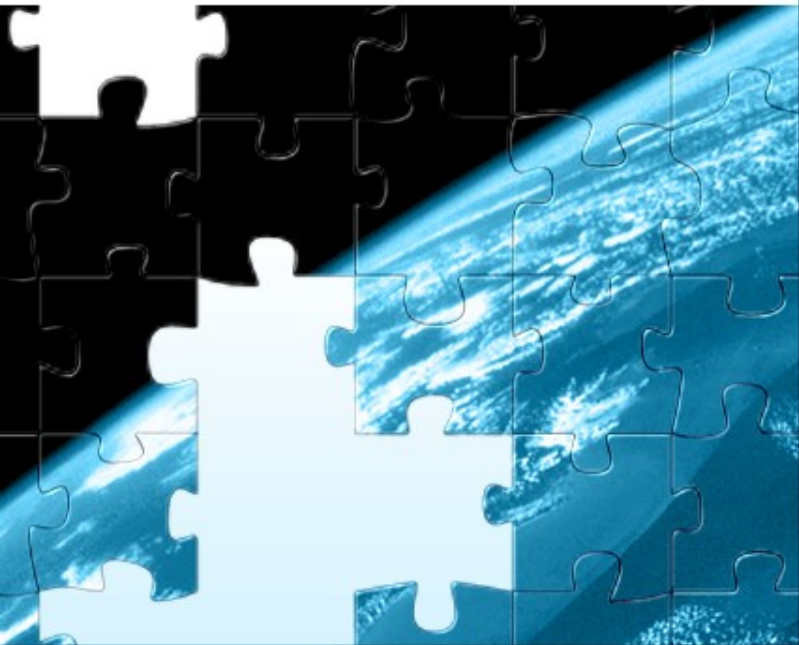
# MySQL Replication Tutorial

Mats Kindahl  
Senior Software Engineer  
Replication Technology

Lars Thalmann  
Development Manager  
Replication/Backup

# Tutorial Outline

- Terminology and Basic Concepts
- Basic Replication
- Replication for scale-out
- Replication for high-availability
- The binary log
- Statements and transactions
- Cluster Replication



Innovation Everywhere

# Terminology and Basic Concepts

# MySQL Replication

## Why?

- **High availability**
  - Fail-over possible
- **Scale-out**
  - Queries on many servers
- **Off-site processing**
  - Do not disturb master
  - Reporting

## How?

### Snapshots

- **Backup**
- **mysqlbinlog**
- **mysqldump**

### Binary log

- **Replication**
- **Point-in-time recovery**

# Terminology

## Master server

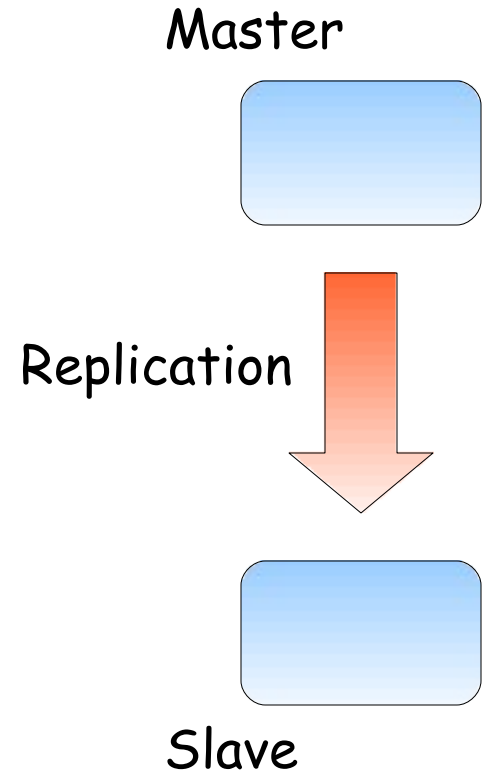
- Changes data
- Keeps log of changes

## Slave server

- Ask master for events
- Executes events

## Binary log

- Log every change
- Split into transactional groups



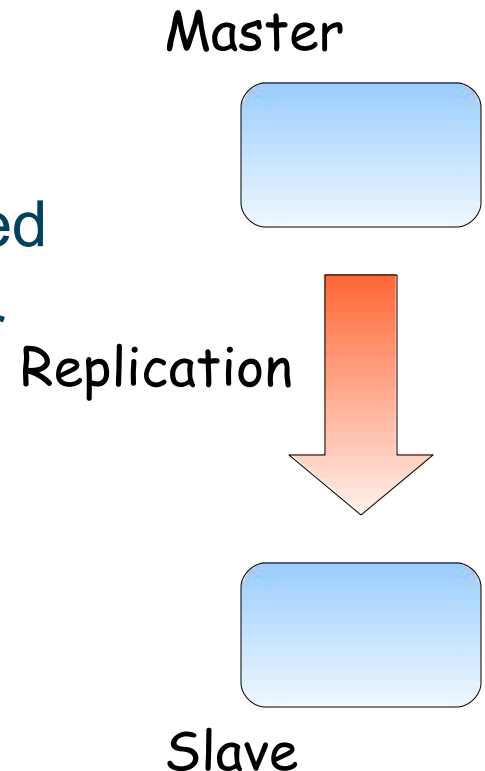
# Terminology

## Synchronous replication

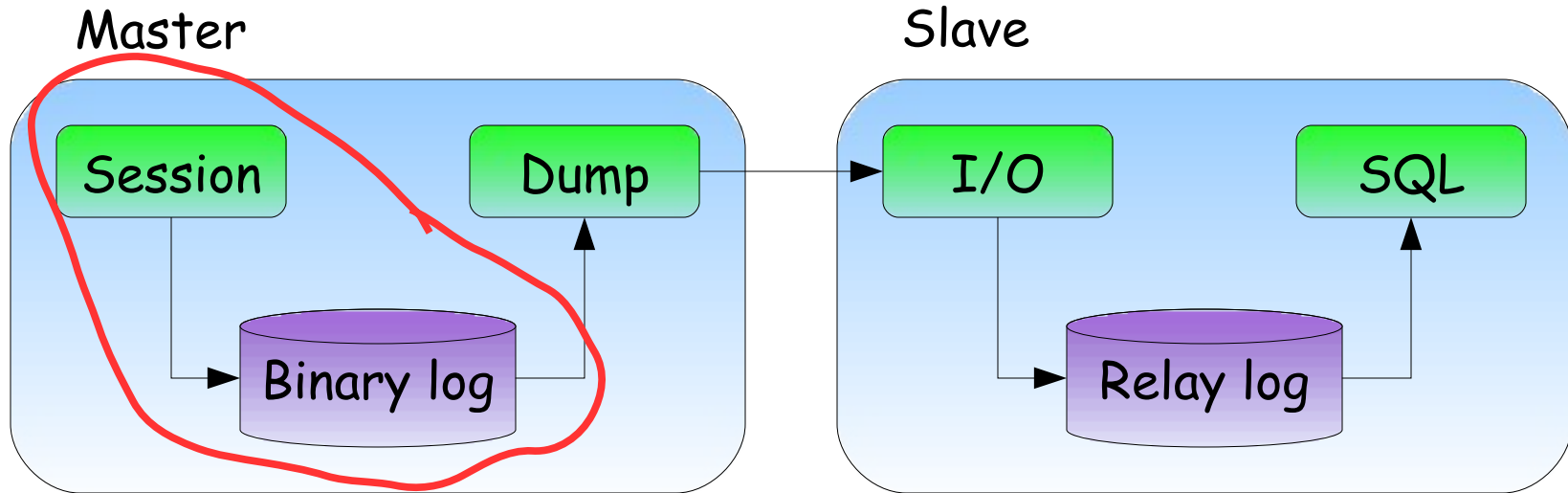
- Transactions are not committed until data is replicated and applied
- Provides consistency, but slower
- Provided by MySQL Cluster

## Asynchronous replication

- Transactions committed immediately and replicated
- No consistency, but faster
- Provided by MySQL Server

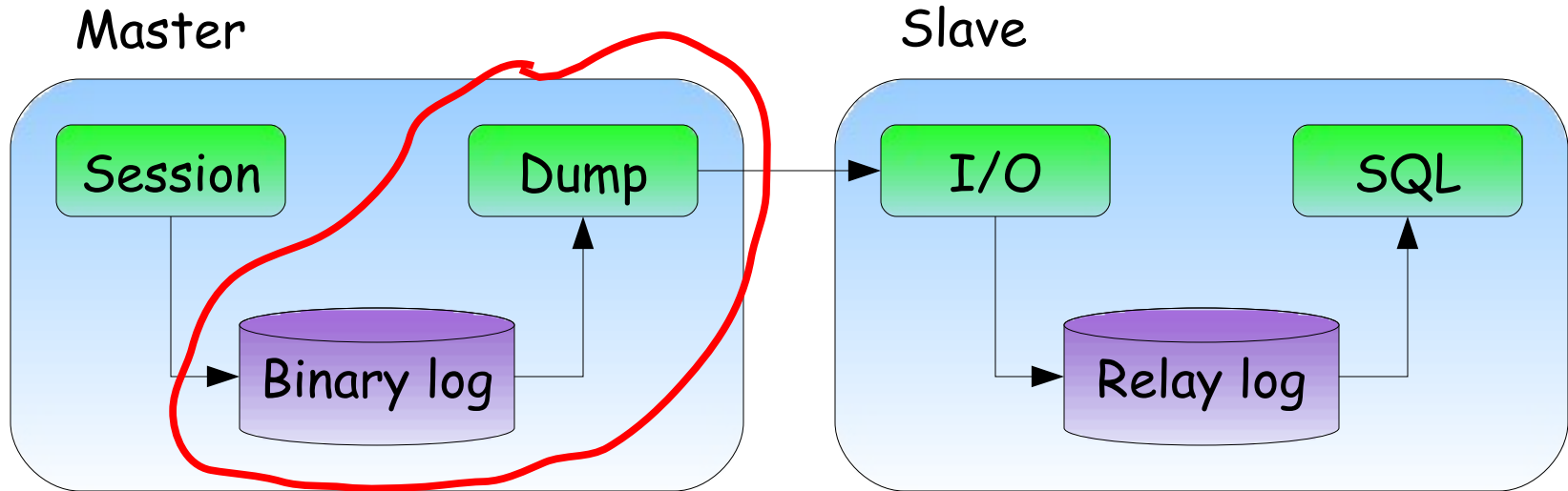


# Replication system architecture



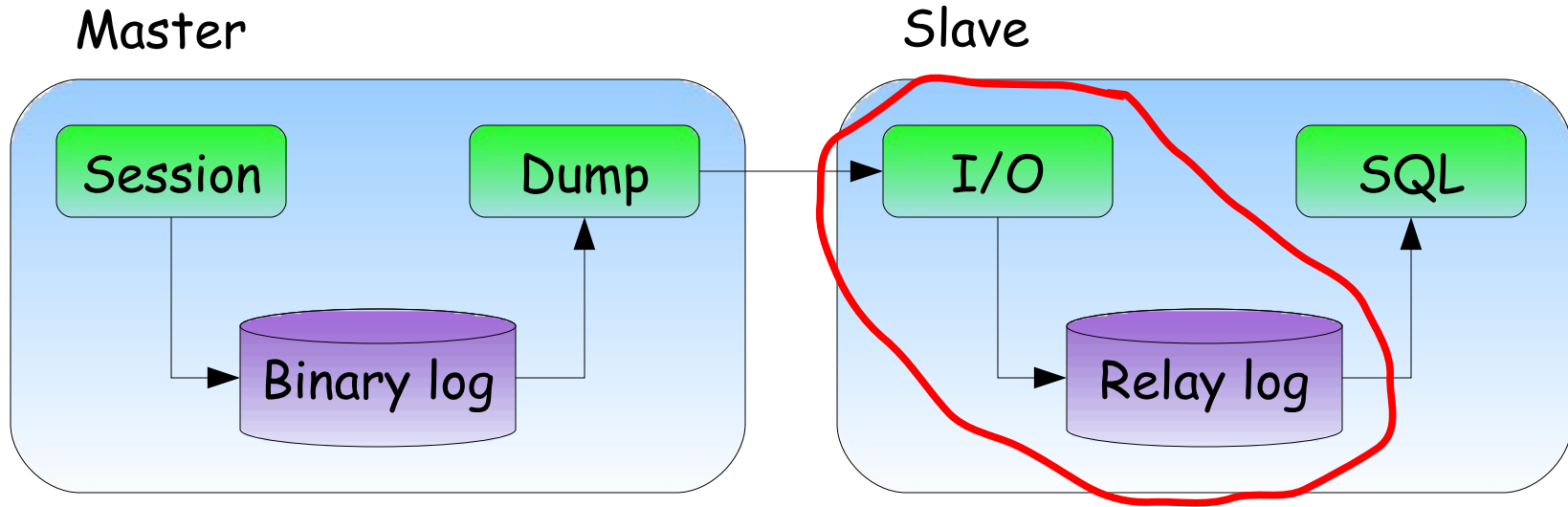
- Binary log (binlog) records all changes
- Consists of transactional *groups*
- Used for replication and point-in-time recovery

# Replication system architecture



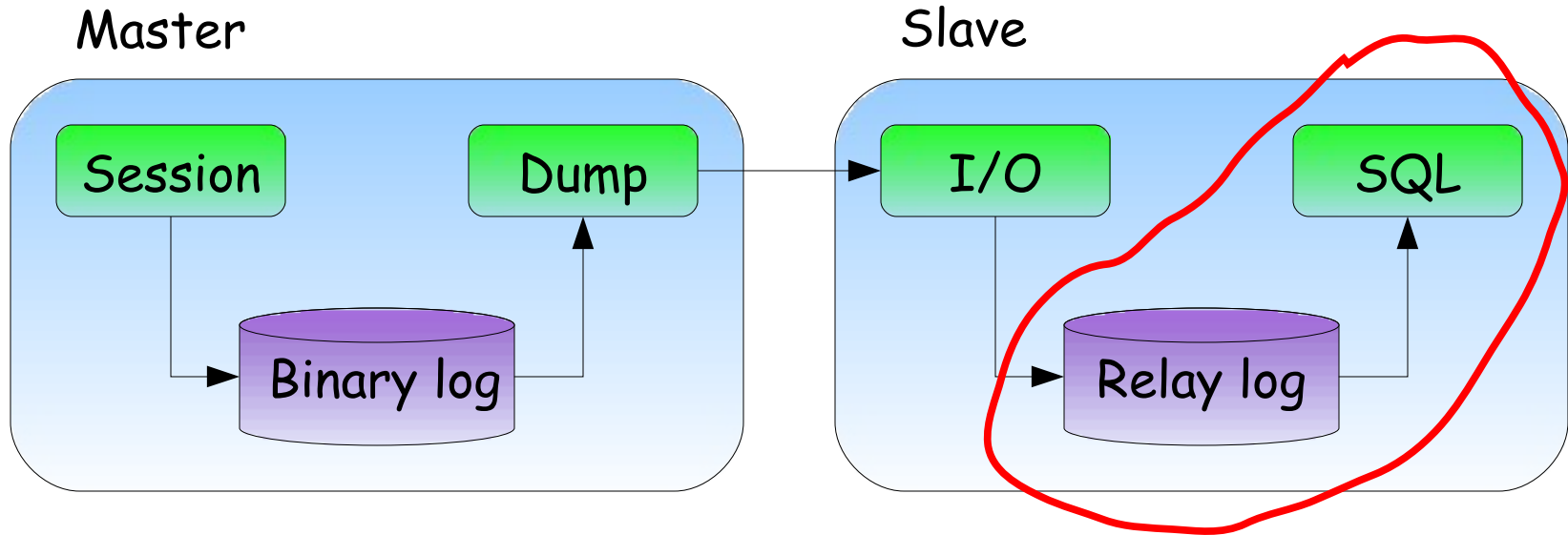
- One dump thread per slave
- Started on request from slave
- Read events from binary log and send to slave

# Replication system architecture



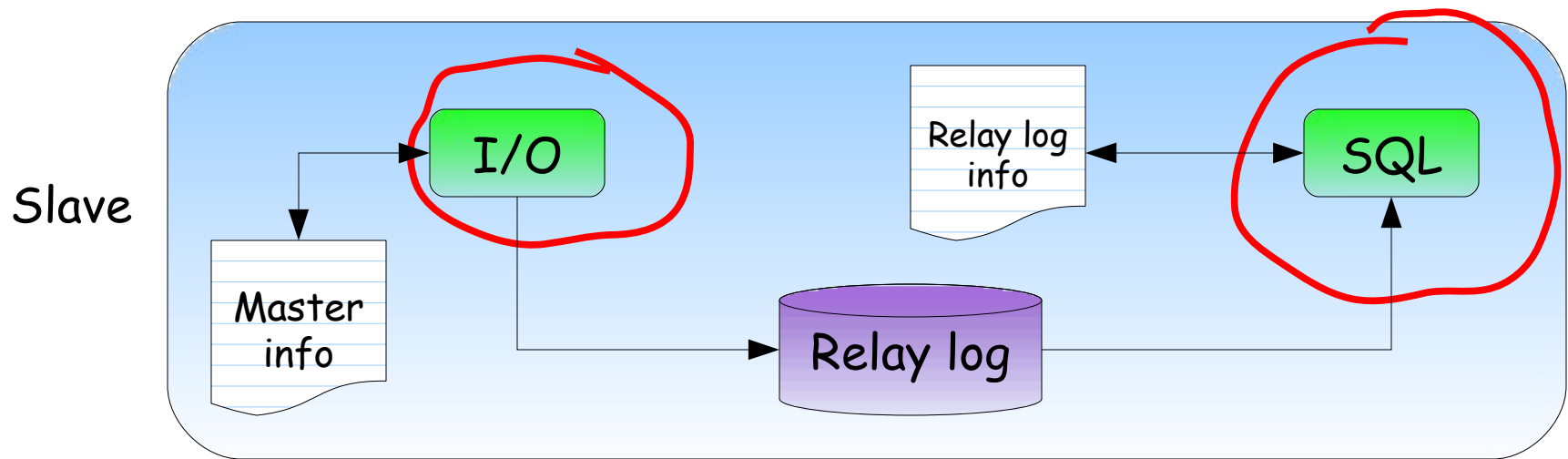
- I/O thread send *dump request* to master
- I/O thread copies events to relay log
- Relay log is a disk-based buffer for events
- Relay log should be considered volatile

# Replication system architecture



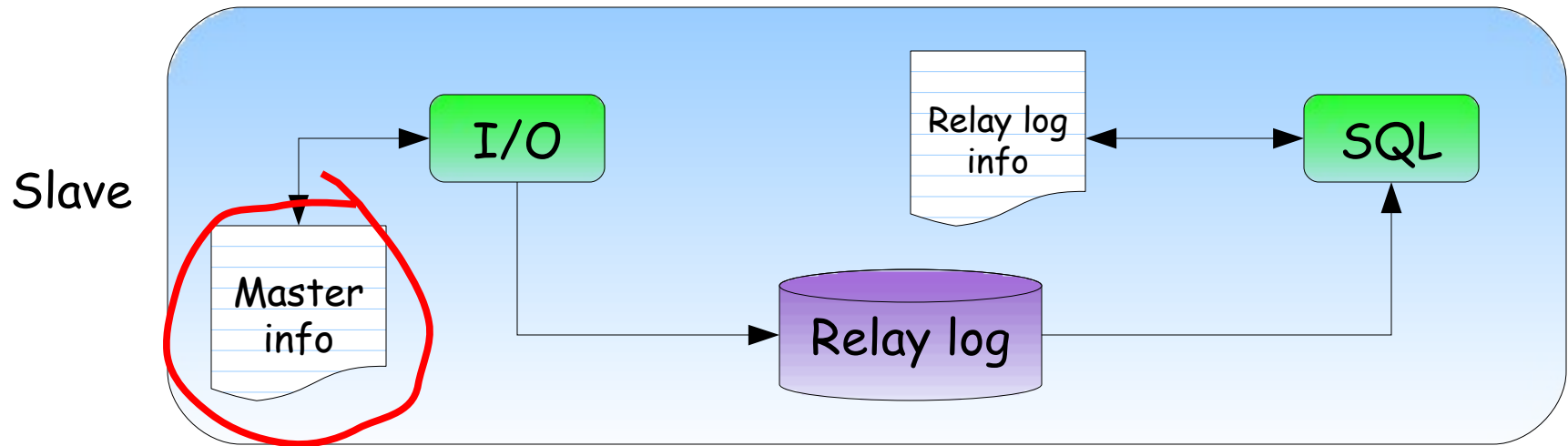
- SQL thread read events from relay log
- SQL thread decodes and applies events to database

# Replication system architecture



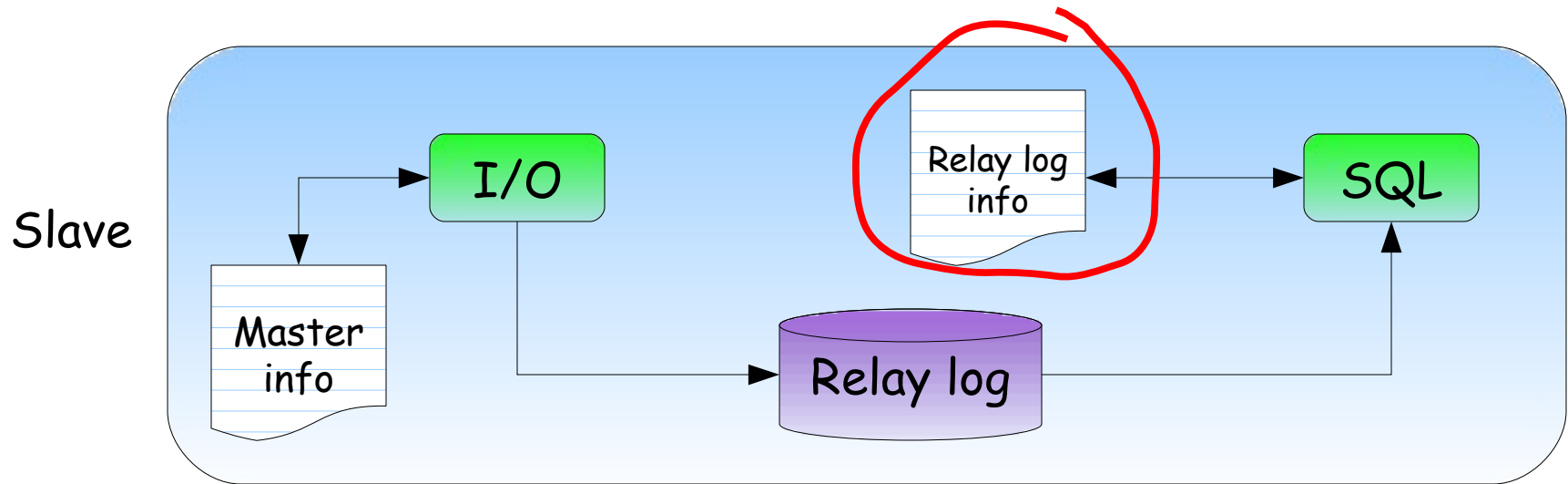
- I/O thread and SQL thread each maintain *binlog coordinates*
- I/O thread maintain *last read (event)*
- SQL thread maintain: *last executed event* and *beginning of last executed group*

# Replication system architecture



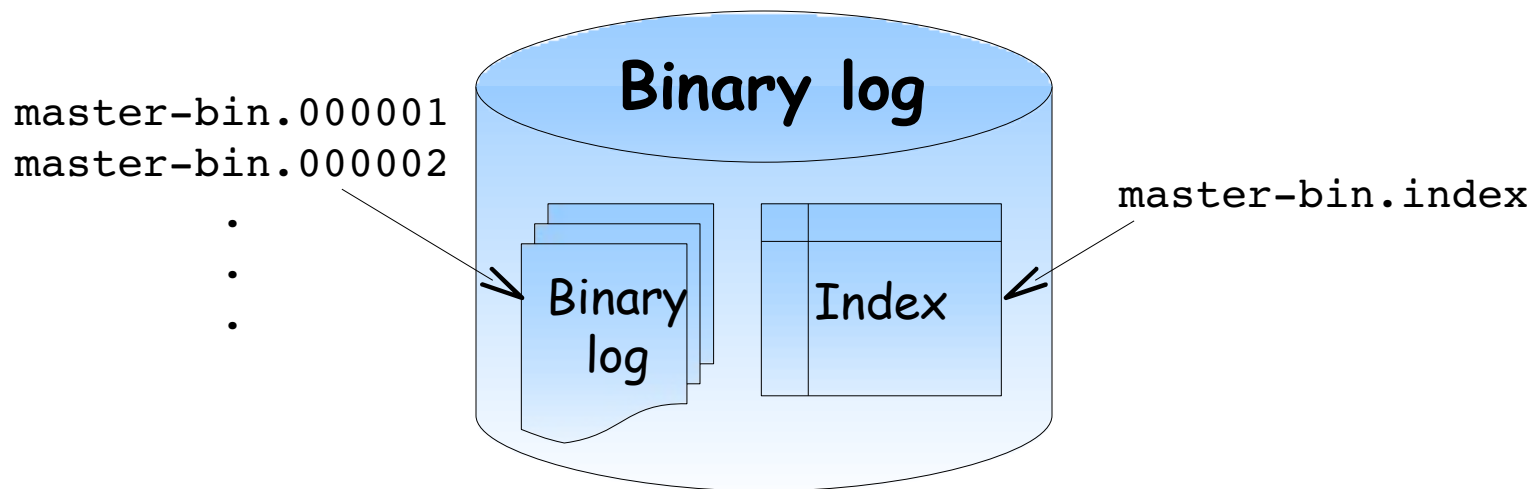
- `master.info` contain:
  - **Read coordinates:** master log name and master log pos
  - **Connection information:**
    - host, user, password, port
    - SSL keys and certificates

# Replication system architecture



- `relay-log.info` contain:
  - **Group master coordinates:**
    - Master log name and master log pos
  - **Group relay log coordinates:**
    - Relay log name and relay log pos

# The binary log



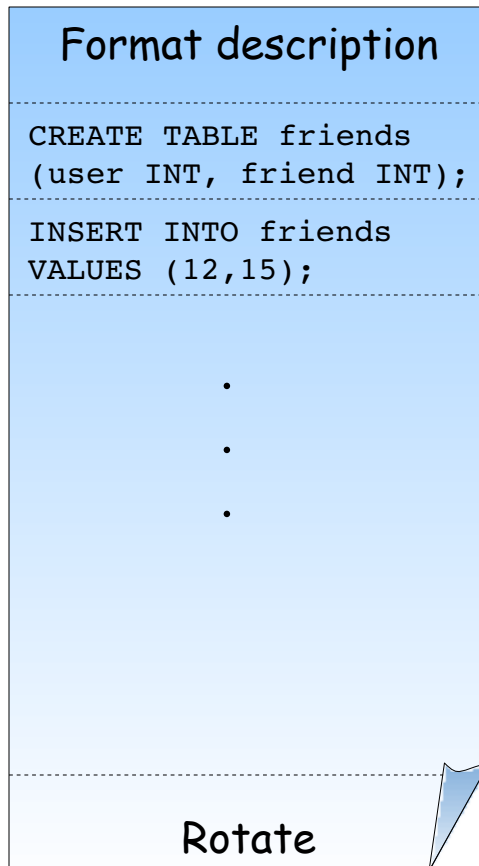
- File: `master-bin.NNNNNN`
  - The actual contents of the binlog
- File: `master-bin.index`
  - An index file over the files above

# A binlog file



Coordinate

master-bin.000001



A binlog event



# Basic Replication

Innovation Everywhere

# Scenario 1: Single slave

- Keep master on-line while:
  - Doing backup
  - Generating reports
  - Adding new slaves

# Scenario 1: Steps

1. Fix my.cnf file for master and slave
2. Add user and grants on master
3. Take backup of master
4. Bootstrap slave from backup
5. Configure slave
6. Start slave

# Step 1: Fix my.cnf

## Master my.cnf

```
[mysqld]
tmpdir = /tmp
language = .../share/english
pid-file = .../run/master.pid
datadir = .../data
server-id = 1
port = 12000
log-bin = .../log/master-bin
socket = /tmp/master.sock
basedir = ...
```

## Slave my.cnf

```
[mysqld]
tmpdir = /tmp
language = .../share/english
pid-file = .../run/slave.pid
datadir = .../data
server-id = 2
port = 12001
socket = /tmp/mysqld.sock
basedir = ...
relay-log-index = ...
relay-log = ...
```

## Step 2: Users and grants

- Create user on master
- Add REPLICATION SLAVE grants

```
master> CREATE USER 'slave_user'@'slave_host';  
master> GRANT REPLICATION SLAVE  
-> ON *.* TO 'slave_user'@'slave_host'  
-> IDENTIFIED BY 'slave_password';
```

## Step 3: Backup master

- Physical backup (offline)
  - For example: `tar`
- Logical backup (offline)
  - `mysqldump`
- On-line backup
  - InnoDB hot backup
  - MySQL on-line backup

## Step 4: Bootstrap slave

- Physical backup
  - Copy backup image to slave
  - Untar into database directory
- Logical backup
  - `mysql` client

## Step 5: Configure slave

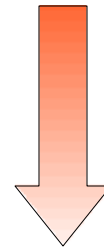
- Use CHANGE MASTER command
  - MASTER\_PORT default is 3306

```
slave> CHANGE MASTER TO
->     MASTER_HOST = 'master_host',
->     MASTER_PORT = 3306,
->     MASTER_USER = 'slave_user',
->     MASTER_PASSWORD = 'slave_password';
```

# Step 6: Start slave!

```
slave> START SLAVE;
```

Master



Slave

# Some suggestions

- Start the binary log on the master immediately following the backup.
  1. Add user and grants on master
  2. Shut down master
  3. Edit my.cnf
  4. Take backup of master
  5. Restart master

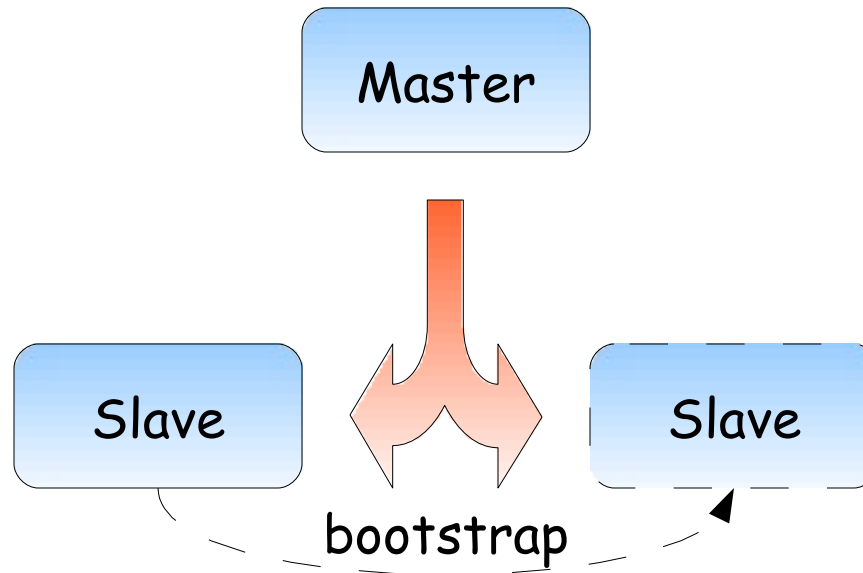
# Some suggestions, contd.

- Deprecated options in my.cnf:
  - `master-host`
  - `master-port`
  - `master-user`
  - `master-password`

# Scenario 1: Summary

- Configure options for my.cnf:
  - `log-bin`
  - `server-id`
- Replication user
  - `GRANT REPLICATION SLAVE`
- Configure slave
  - `CHANGE MASTER TO`
- Starting slave
  - `START SLAVE`

# Scenario 2: Add new slave



- Bootstrap from slave (Backup)
- Start from coordinate of Backup
- Master does not have to be stopped

# Adding a slave

1. Stop existing slave
2. Take note of stop position
3. Backup existing slave
4. Start existing slave
5. Bootstrap new slave:
  - Fix my.cnf
  - Restore backup
  - Configure new slave
6. Start new slave

# Step 1: Stop slave

- Bring slave off-line

```
slave-1> STOP SLAVE;
```

## Step 2: Note position

- Take a note of where slave stopped
- We need this when starting new slave

```
slave-1> SHOW SLAVE STATUS;
```

```
...
```

```
Relay_Master_Log_File: master-bin.000001
```

```
...
```

```
Exec_Master_Log_Pos: 409
```

# Step 3: Backup slave

- Flush tables and lock database

```
FLUSH TABLES WITH READ LOCK
```

- Take backup

```
tar zcf slave-backup.tar.gz ...
```

- Unlock database

```
UNLOCK TABLES
```

# Step 4: Start slave

- We can now start slave again since:
  - We have the master position of the slave
  - We have a backup corresponding to that position

```
slave-1> START SLAVE;
```

# Step 5: Bootstrap new slave

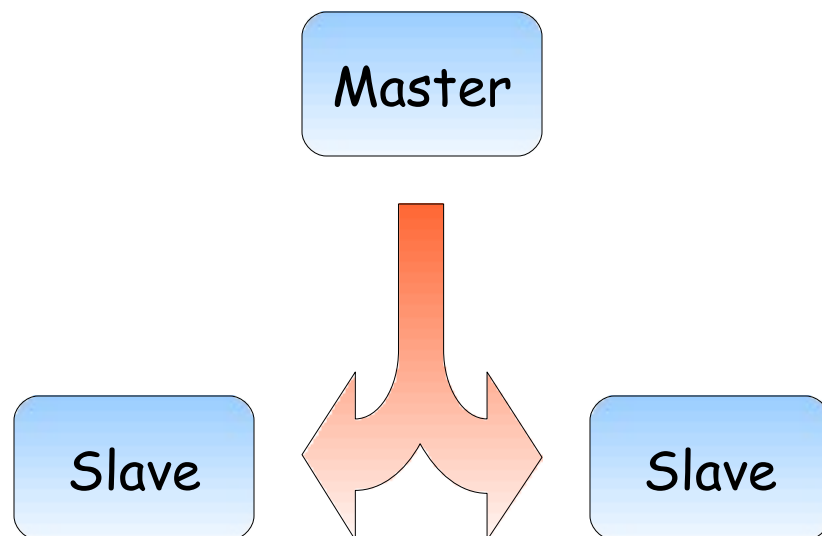
- Fix my.cnf (use new server-id!)
- Install backup  
`tar xcf slave-backup.tar.gz ...`
- Configure slave using saved position

```
slave-2> CHANGE MASTER TO
->     MASTER_HOST = 'master_host',
->     MASTER_PORT = 3306,
->     MASTER_USER = 'slave_user',
->     MASTER_PASSWORD = 'slave_password',
->     MASTER_LOG_POS = 409,
->     MASTER_LOG_FILE = 'master-bin.000001';
```

# Step 6: Start new slave

- Start new slave!
- It will start from the position corresponding to the backup

```
slave-2> START SLAVE;
```



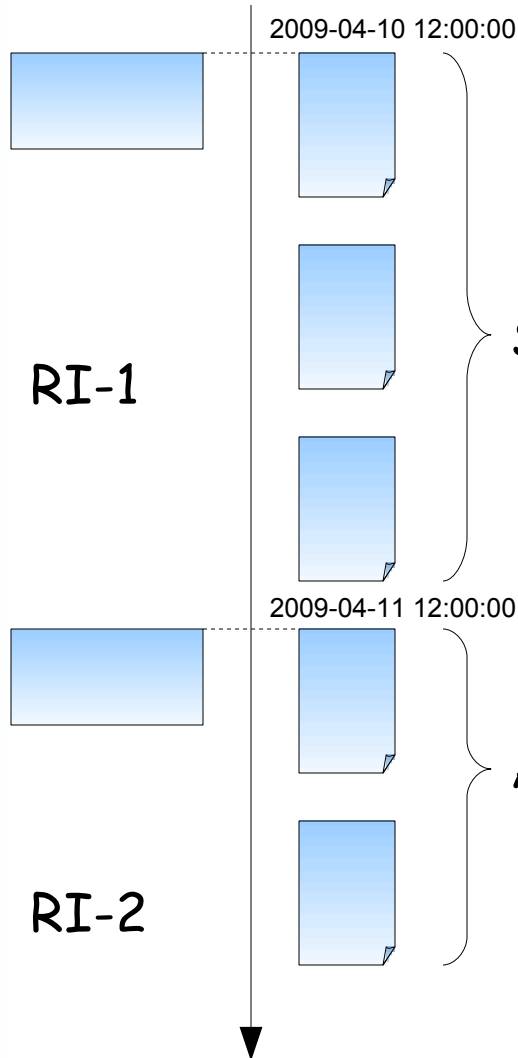
# Scenario 2: Summary

- Taking a snapshot of a slave
  - STOP SLAVE
  - FLUSH TABLES WITH READ LOCK
  - SHOW SLAVE STATUS
  - UNLOCK TABLES
- Starting replication from anywhere
  - MASTER\_LOG\_FILE
  - MASTER\_LOG\_POS

# Scenario 3: Point-in-time recovery

- Binlog for point-in-time recovery
  - Say: time T
- Backup needs to:
  - Save backup image
  - Save binlog files
- Recover needs to:
  - Restore backup image
  - Apply binlog files until T

# Recovery images



- Backup image + Binlog files = *Recovery image*

Saved recovery image

- Saved recovery images can be archived

Active recovery image

- Active recovery image is still growing

# Scenario 3: Backup steps

RI-n

## 1. Lock database

FLUSH TABLES WITH READ LOCK

## 2. Note binlog file and position

SHOW MASTER STATUS

## 3. Backup server

## 4. Save previous recovery image

## 5. Unlock database

UNLOCK TABLES

# Saving recovery image

RI-n

- We are starting image RI-n
- Save away binlogs for image RI-(n-1)
  - **Start:** positions for image RI-(n-1)
  - **End:** positions for image RI-n
- Keep track of active recovery image:
  - Backup image for RI-n
  - Positions when image RI-n started

# Scenario 3: Recovery steps

1. Check if active recovery image  
Last  $\leq$  Target
2. Find correct saved recovery image  
Start  $\leq$  Target < End
3. Restore backup image
4. Apply binary logs to date

```
mysqlbinlog \  
  --start-position=position \  
  --stop-datetime=target \  
  master-bin.000022 ... master-bin.000028
```

# Scenario 3: Summary

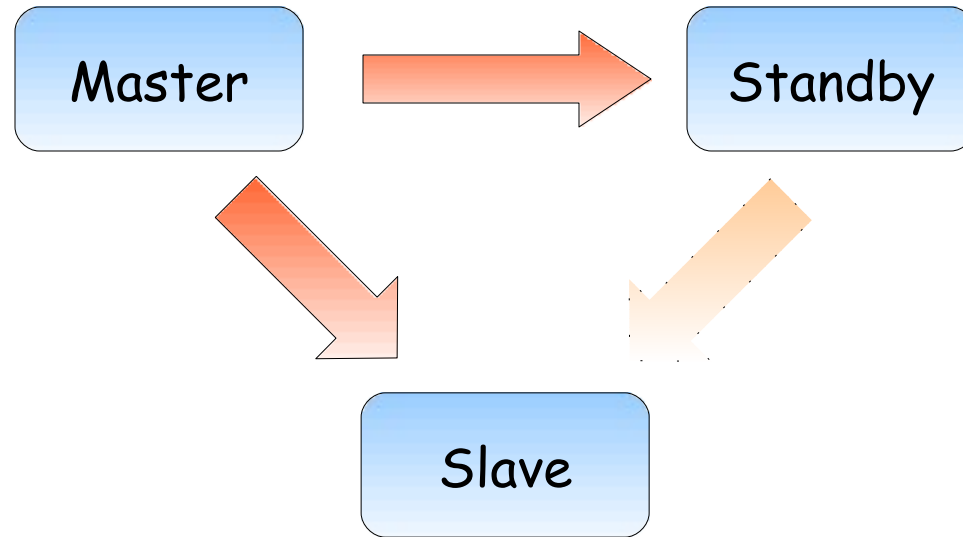
## Backup:

- Note binlog position for each backup
- Archive binlog files with backup

## Restore:

- Restore backup image
- Use `mysqlbinlog` to “play” binlogs

# Scenario 4: Standby master



- Use slave as standby master
- Bring master down for maintenance

# Switch over to standby

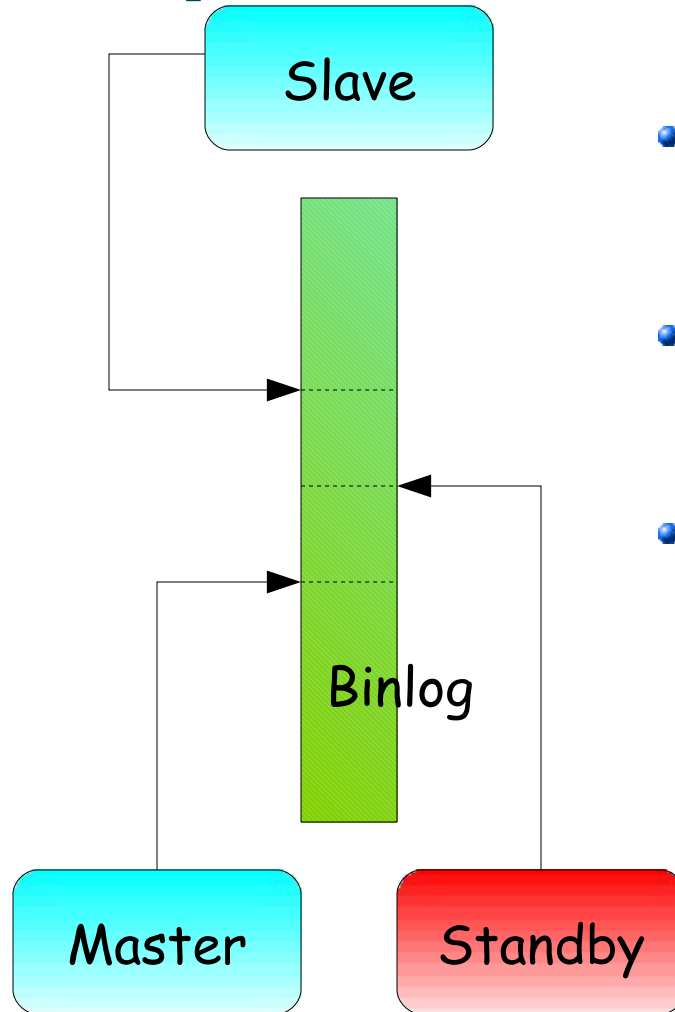
1. Configure standby master
2. Ensure standby is ahead of slave
3. Stop standby (and bring it offline)
4. Note master and standby position
5. Bring slave to standby position
6. Redirect slave to standby
7. Start slave

# Step 1: Configure standby

- Configure standby to log replicated events
  - `log-slave-updates`

```
[mysqld]  
...  
log-slave-updates
```

# Step 2: Standby ahead



- Standby **have** to be ahead of slave
- Standby have to be “more knowledgeable”
- Nothing to replicate otherwise

## Step 2: Standby be ahead

- Pick the “most knowledgeable” slave as standby
- Do this fandango:
  - Stop slave
  - Note master position M
  - Stop standby
  - Start standby until M
  - Wait for standby to reach M
- Standby will now have stopped

## Step 2: Standby be ahead

- Commands for doing this

```
slave> STOP SLAVE;
```

```
slave> SHOW SLAVE STATUS;
```

```
Relay_Master_Log_File: master-bin.00032
```

```
Exec_Master_Log_Pos: 567823
```

```
standby> STOP SLAVE;
```

```
standby> START SLAVE UNTIL
```

```
-> MASTER_LOG_FILE = 'master-bin.00032',
```

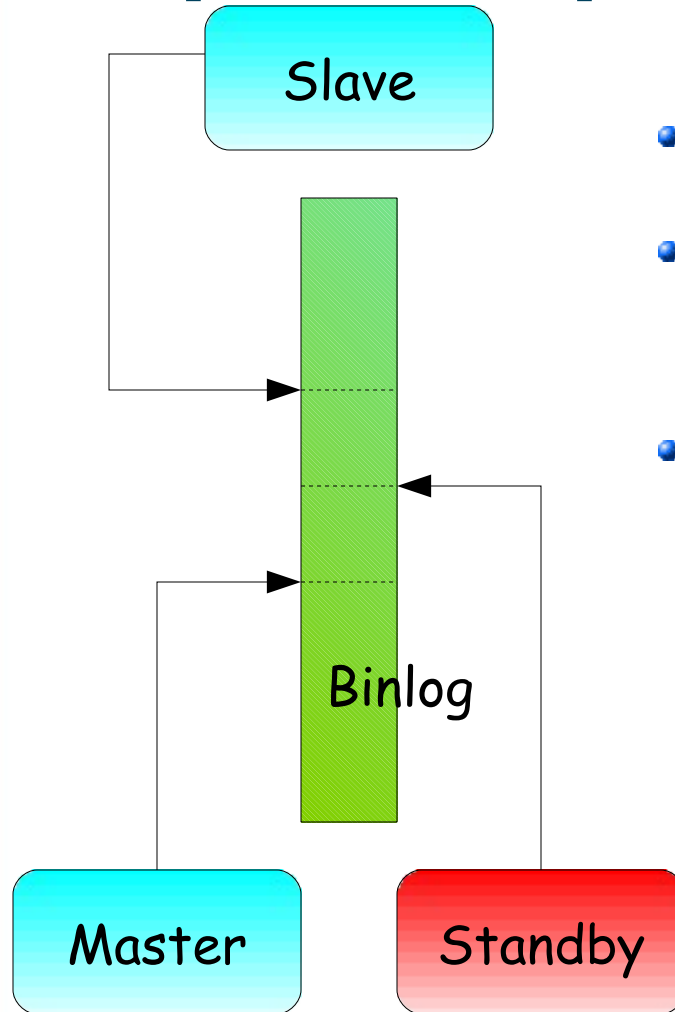
```
-> MASTER_LOG_POS = 567823;
```

```
standby> SELECT
```

```
-> MASTER_POS_WAIT('master-bin.00032',
```

```
-> 567823);
```

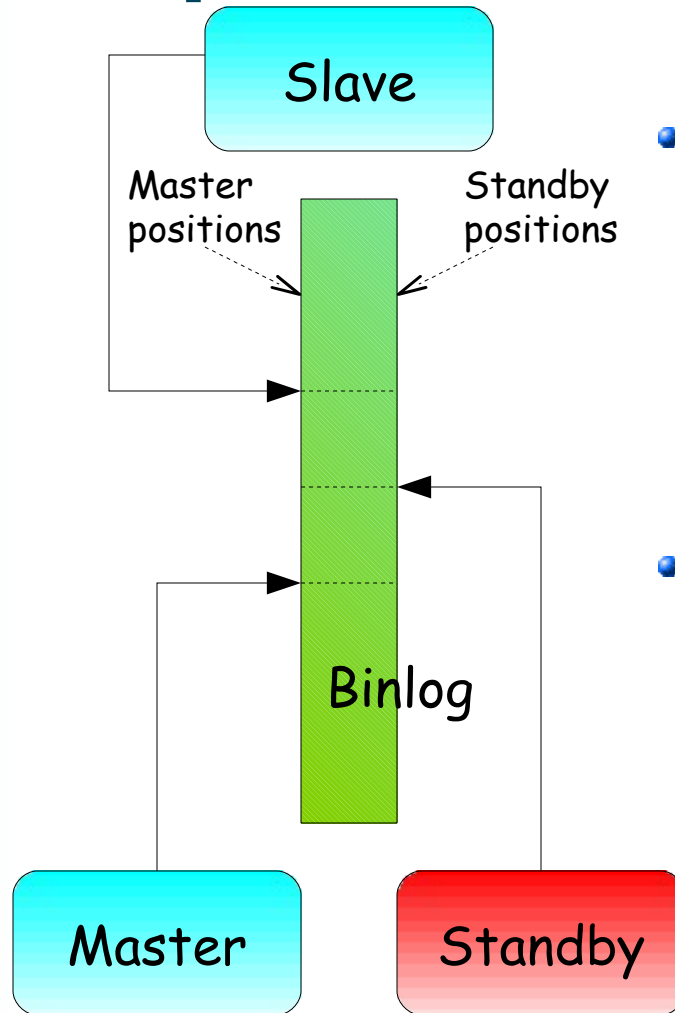
# Step 3: Stop standby



- Stop standby
- Slave is already stopped
- Optional: bring standby off line

FLUSH TABLES  
WITH READ LOCK

# Step 4: Standby positions



- Standby have *two* positions
  - Master position
  - Standby position
- Need to match master position to standby position

# Step 4: Master position

- Note master position of where standby stopped
- Same as before

```
standby> SHOW SLAVE STATUS;
```

```
...
```

```
Relay_Master_Log_File: master-bin.000032
```

```
...
```

```
Exec_Master_Log_Pos: 7685436
```

# Step 4: Standby position

- Note of last binlogged event
- No changes allowed on server!

```
standby> SHOW MASTER STATUS\G
```

```
***** 1. row *****
```

```
File: standby-bin.000047
```

```
Position: 7659403
```

```
Binlog_Do_DB:
```

```
Binlog_Ignore_DB:
```

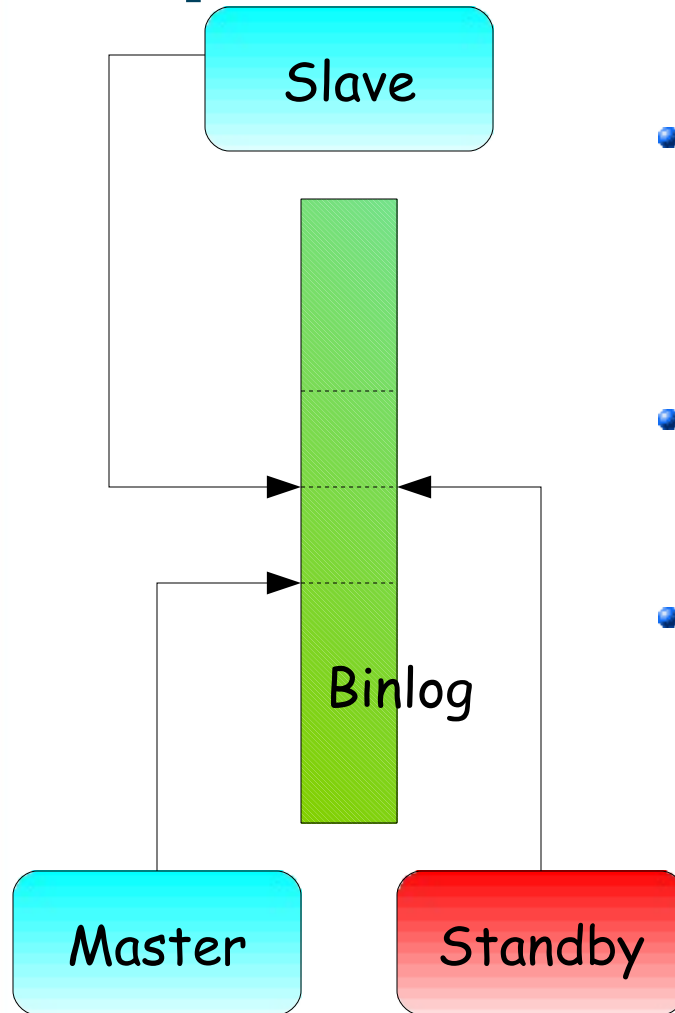
```
1 row in set (0.00 sec)
```

# Step 5: Start slave until

- We now have:
  - A binlog position on the master
  - A binlog position on the standby
- Optional: bring standby on-line  
**UNLOCK TABLES**
- Run slave until master position

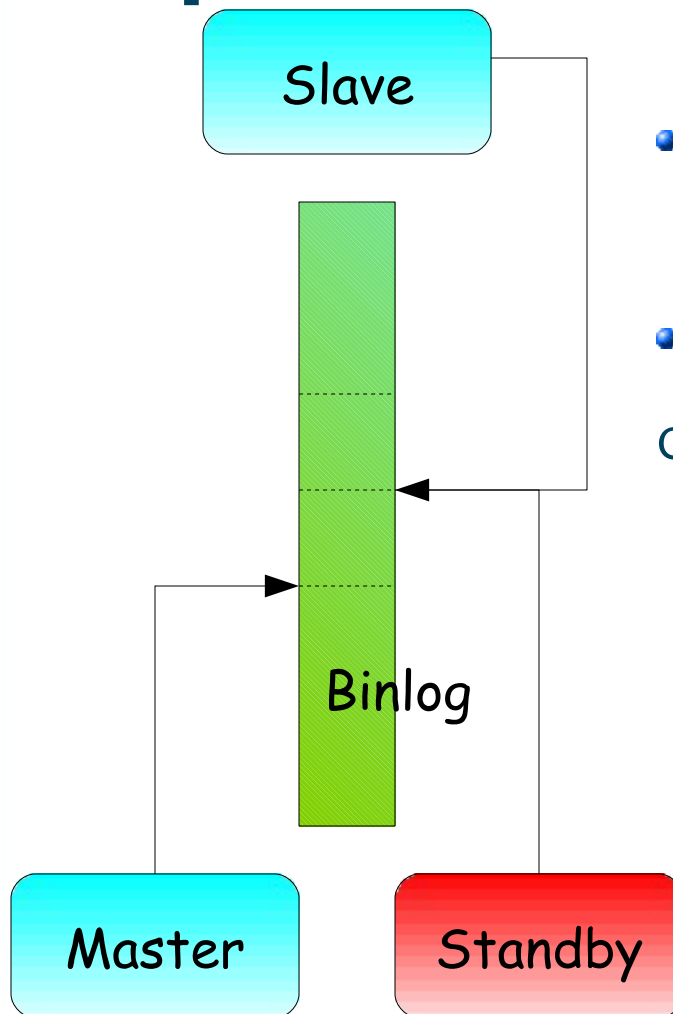
```
slave> START SLAVE UNTIL  
-> MASTER_LOG_FILE = 'master-bin.000032',  
-> MASTER_LOG_POS = 7685436;
```

# Step 6: Redirect slave



- Slave stopped at master binlog position
- Standby stopped at the same position
- You know the standby position

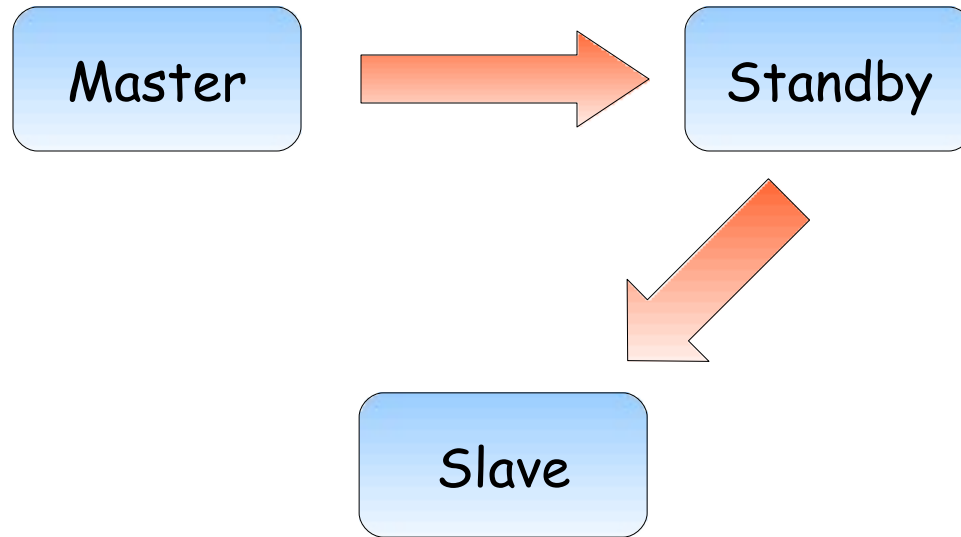
# Step 6: Redirect slave



- Redirect slave to standby position
- Use standby position

```
CHANGE MASTER TO  
MASTER_HOST = ...,  
MASTER_PORT = ...,  
MASTER_LOG_FILE =  
  'standby-bin.000047',  
MASTER_LOG_POS =  
  7659403;
```

# Step 7: Start slave

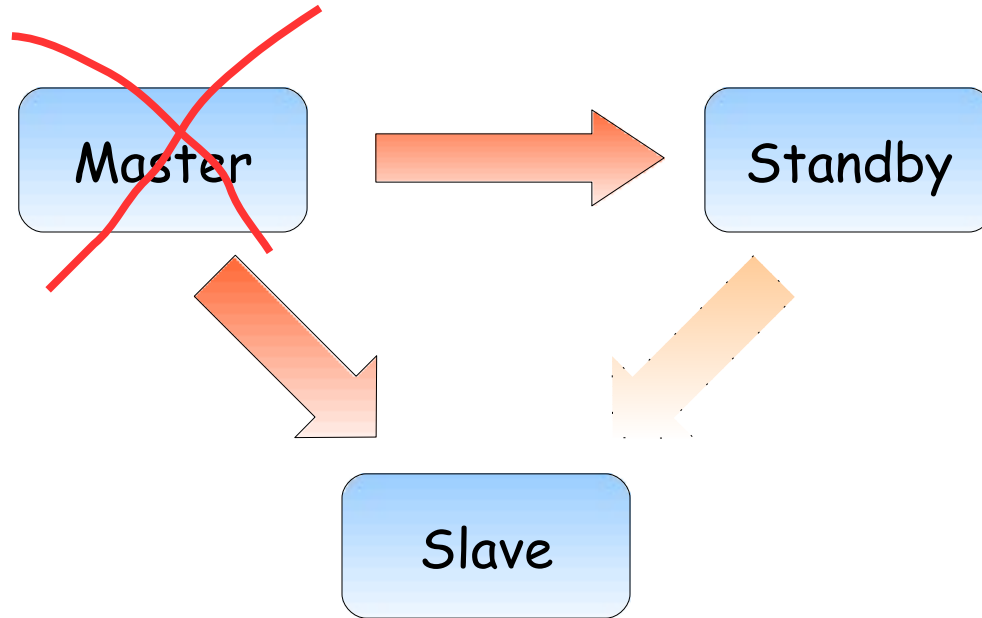


```
slave> START SLAVE;
```

# Scenario 4: Summary

- Forwarding replication events
  - `log-slave-updates`
- Standby **have** to be ahead of Slave
  - ... and ways to ensure that
- Synchronizing for switch-over
  - `SHOW MASTER STATUS`
  - `START SLAVE UNTIL`
  - `MASTER_POS_WAIT( )`

# What about crashes?



- Not possible to check master
- Pick “most knowledgeable” slave:
  - Query each slave
  - Redirect other slaves

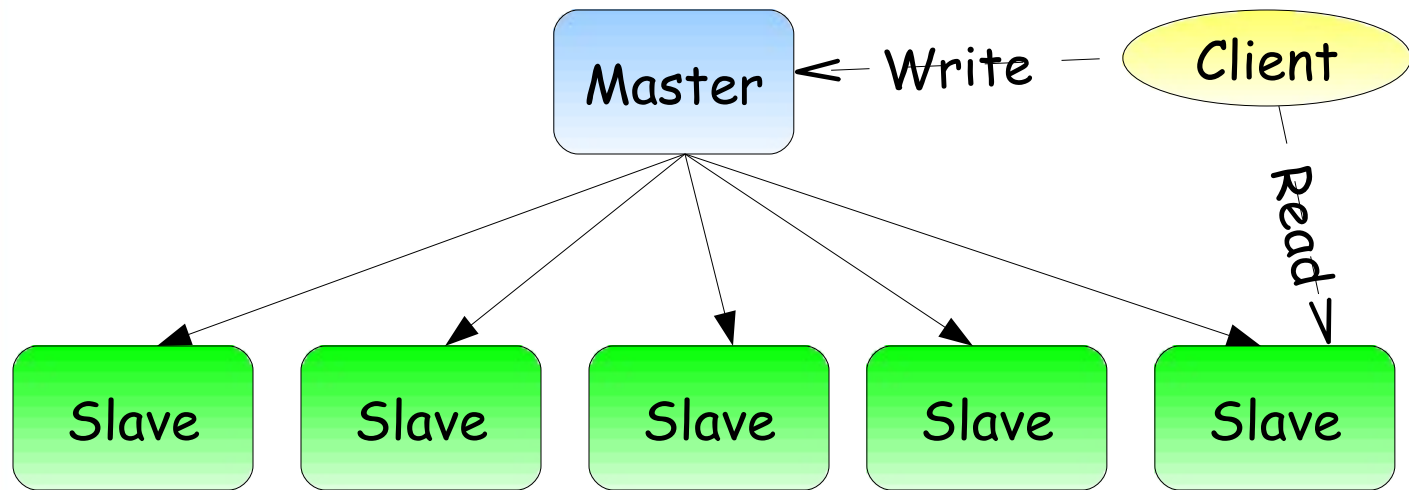


# Replication for Scale-out

Keeping the system  
responsive

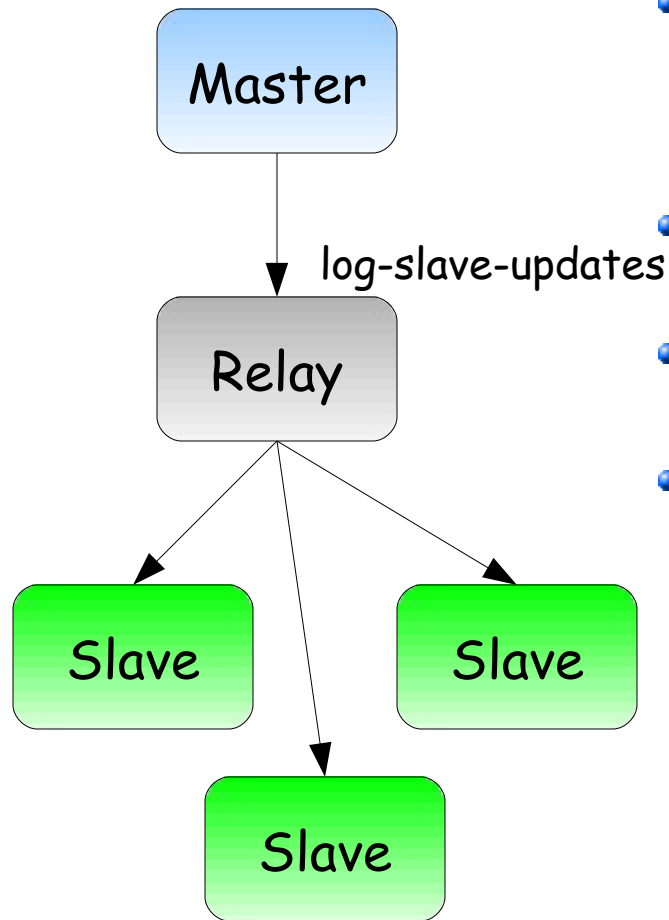
Innovation Everywhere

# Scaling out



- Distribute *read* query processing
- *Write* queries still go to master
- Clients need to send:
  - Read queries to a slave
  - Write queries to the master

# Scenario 5: Relay slave



- Reduce load on master
- Binary log on relay
- No tables on relay
- **BLACKHOLE**

# Scenario 5: Relay slave

1. Stop slave
2. Change default storage engine
3. Change engine of existing tables
4. Start slave

# Step 2: Change engine

- Change default engine on relay

```
SET GLOBAL  
    STORAGE_ENGINE = 'BLACKHOLE';
```

- New tables will use BLACKHOLE

# Step 3: Change engine

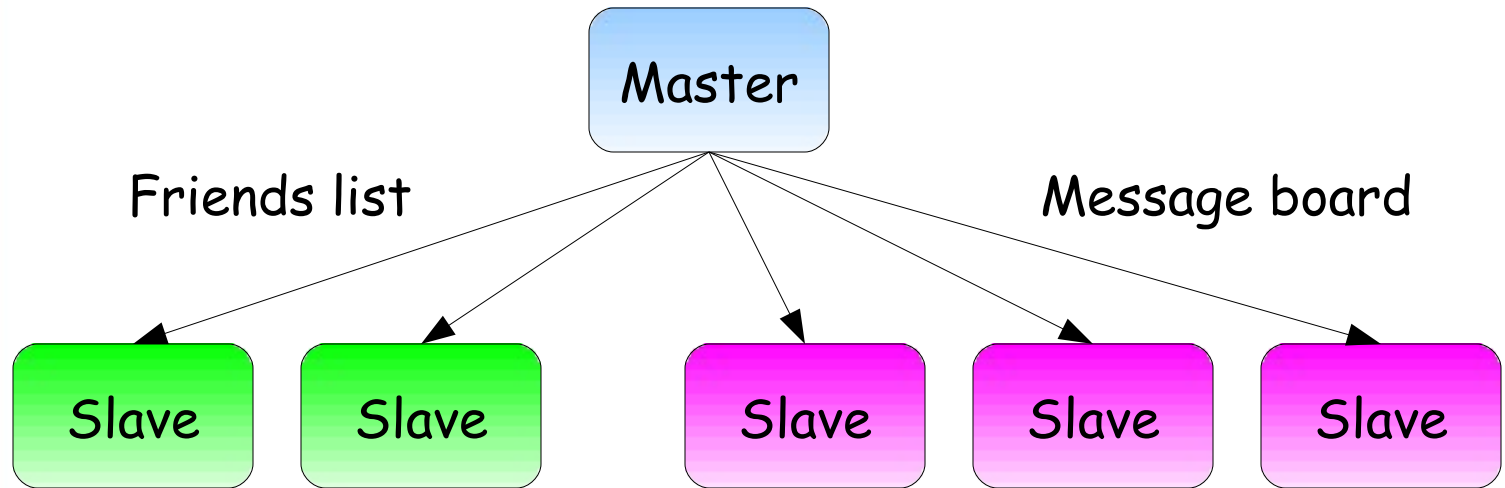
- Change engine for existing tables
  - ... should not be logged
  - So we turn of logging

```
SET SQL_LOG_BIN = 0;  
ALTER TABLE table  
    ENGINE = BLACKHOLE;  
SET SQL_LOG_BIN = 1;
```

# Scenario 5: Summary

- Use BLACKHOLE engine
- Change default engine  
`SET GLOBAL STORAGE_ENGINE=engine`
- Change engine of existing tables  
`ALTER TABLE ENGINE=engine`

# Scenario 6: Specialist slaves



- Scale out dependent on role
- Only store tables that are needed
  - Remove other tables
  - Need to filter out changes

# Scenario 6: Adding filters

1. Shutdown server
2. Edit `my.cnf` file to add filters
3. Restart server

There are:

- Master filtering
- Slave filtering

# Step 2: Edit my.cnf

```
[mysqld]
...
replicate-do-table=user
replicate-do-table=friend
```

Friends slave

```
[mysqld]
...
replicate-do-table=user
replicate-do-table=message
```

Message board slave

- Add slave filtering rules to my.cnf
- Multiple options for multiple rules

# Master side filtering rules

- Filtering on database
- Filtered events not in binary log
  - No point-in-time recovery

- Filtering rules:

`binlog-do-db`

`binlog-ignore-db`

# Slave side filtering rules

- Filter on database, table, or pattern
- Events read from relay log  
... but not executed
- Filtering rules:
  - `replicate-do-db`
  - `replicate-ignore-db`
  - `replicate-do-table`
  - `replicate-ignore-table`
  - `replicate-wild-do-table`
  - `replicate-wild-ignore-table`

# Filtering notes

- Either `*-ignore-db` or `*-do-db`
  - `*-ignore-db` ignored otherwise
- Statements are filtered based on *current database*

- Filtered:

```
USE filtered_db;  
INSERT INTO plain_db.tbl ...
```

- Not filtered

```
USE plain_db;  
INSERT INTO filtered_db.tbl ...
```

# Scenario 6: Summary

- Filtering rules added to `my.cnf`  
... requires server shutdown

- Master filtering

`binlog-do-db`, `binlog-ignore-db`

- Slave filtering

`replicate-do-db`, `replicate-ignore-db`

`replicate-do-table`

`replicate-ignore-table`

`replicate-wild-do-table`

`replicate-wild-ignore-table`

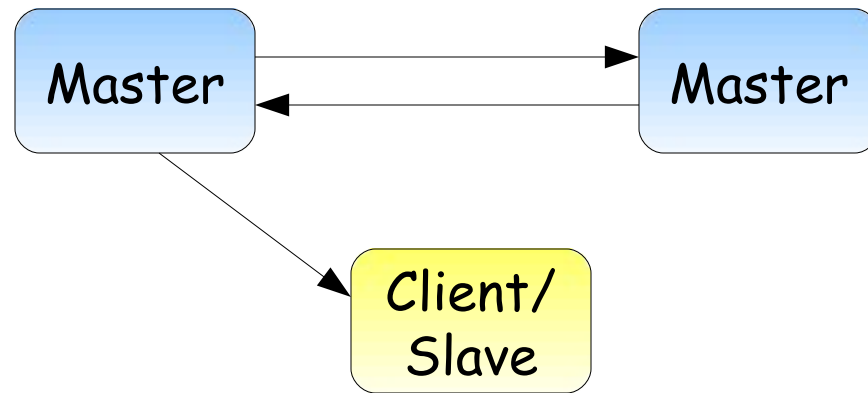


# Replication for High- Availability

Keeping them servers up  
and running

Innovation Everywhere

# Scenario 7: Dual masters



- High-availability
- One master can fail
- *Not* scale-out

# Scenario 7: Dual masters

## 1. Configure masters as slaves

- `server-id`
- `log-bin`
- Add user and grants

## 2. For scale-out usage:

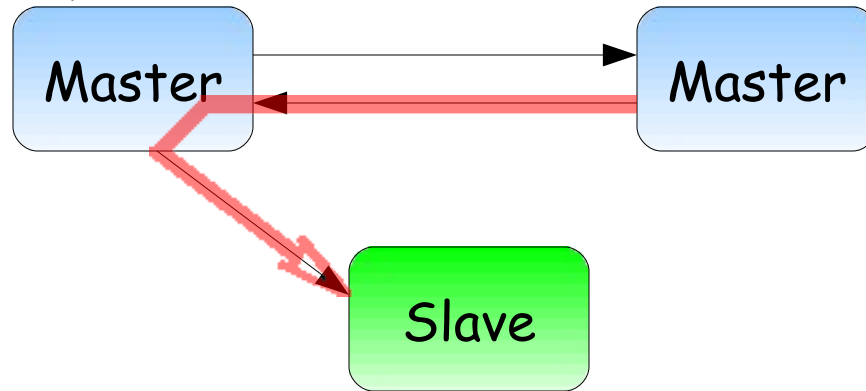
- `log-slave-updates`

## 3. Direct masters to each other

- `CHANGE MASTER TO`
- `START SLAVE`

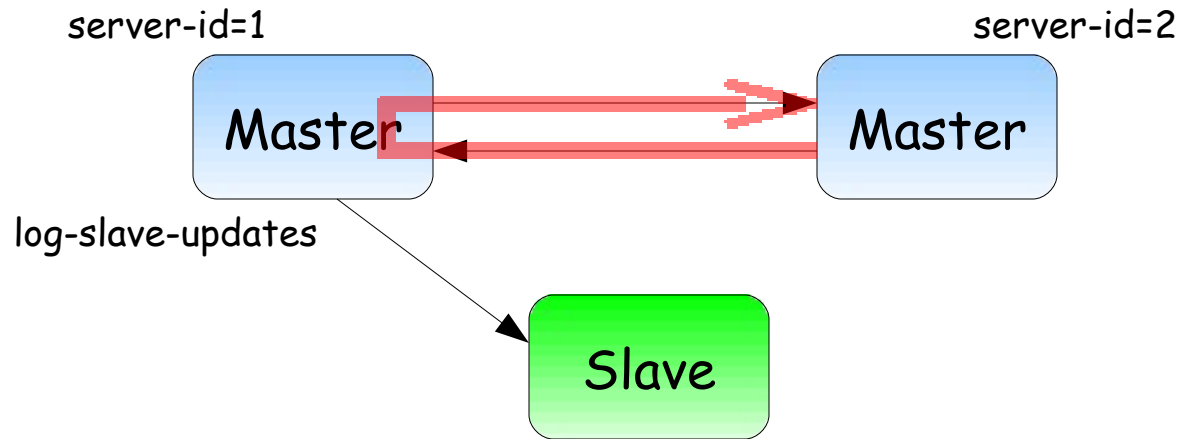
# log-slave-updates?

log-slave-updates



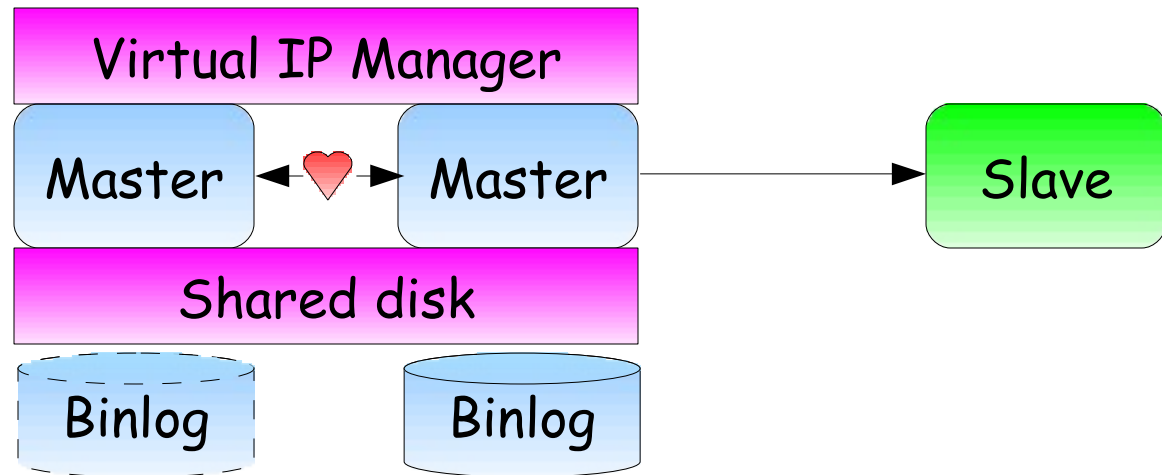
- Use `log-slave-updates`?
  - Necessary to forward events
- Consider: recovery?
- Consider: connecting a slave later?

# Events coming back?



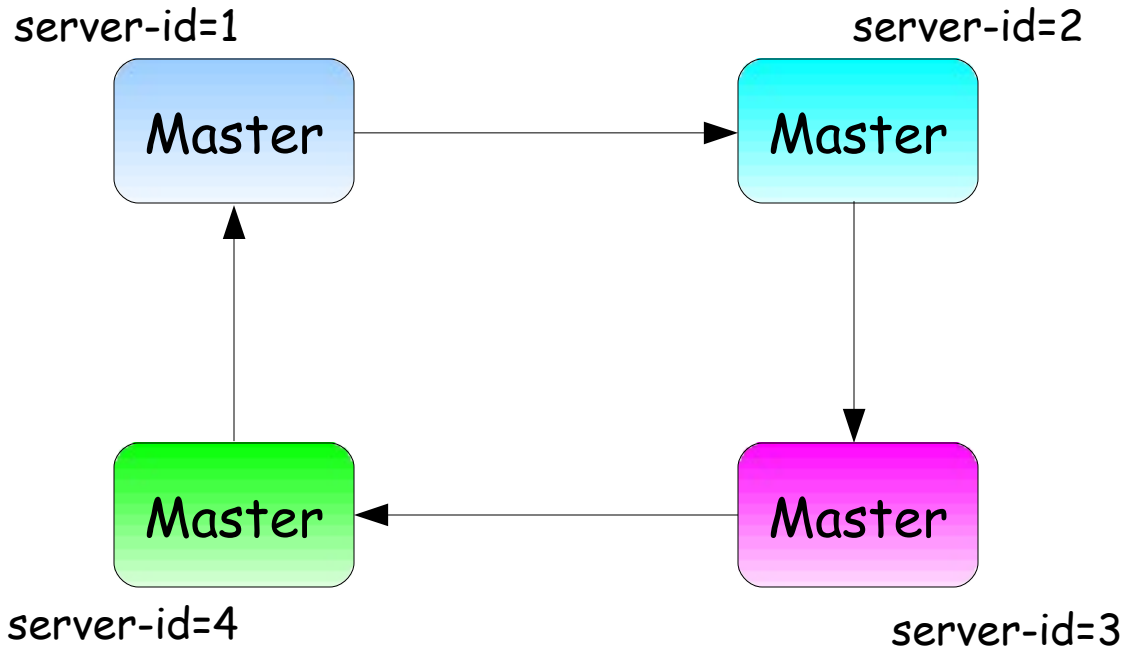
- Master is also a slave
  - Will see its own events
- Server id is stored in event
- Same server id is filtered
  - `replicate-same-server-id`

# Shared disk



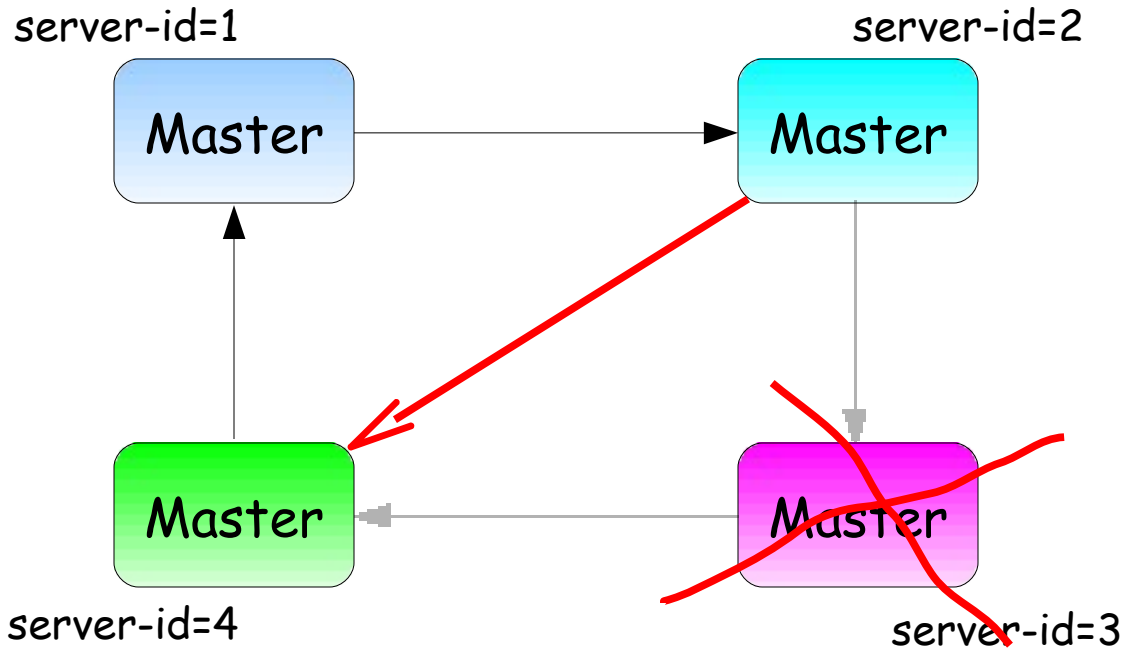
- Active/Passive pair
- Master and slave share binlog
  - Shared store: DRBD, RAID
  - On fail-over, binlog positions match

# Circular replication?



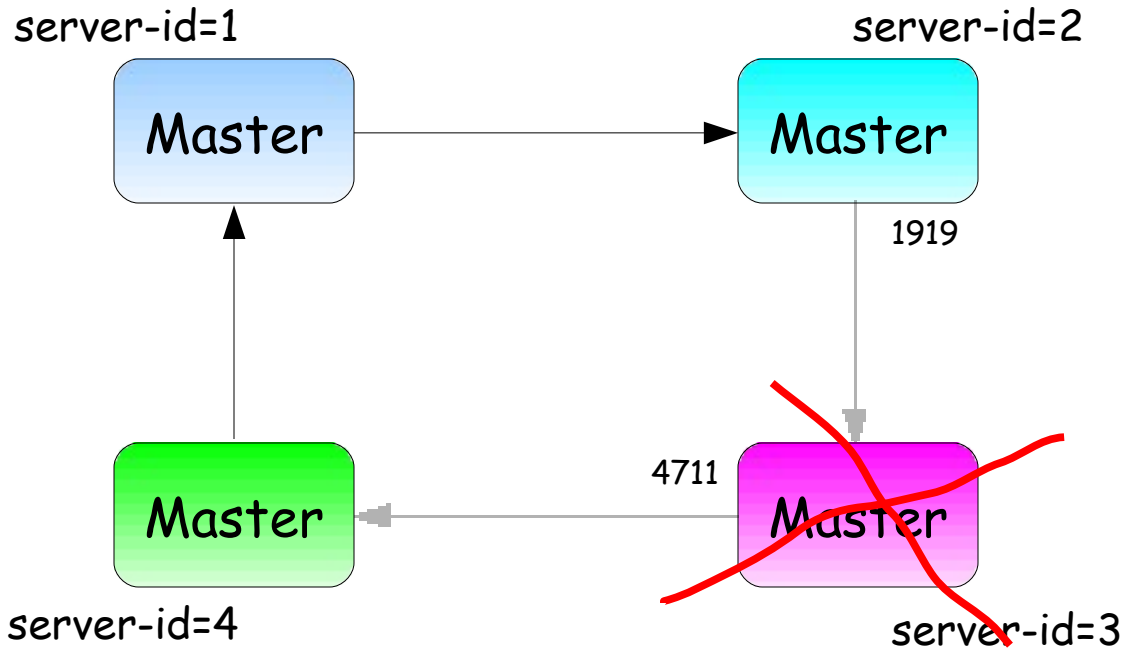
- Replicate in a ring
- Not a recommended setup
  - Complicated to maintain

# Circular replication?



- What if one master crashes?
  - Need to “shrink” ring
  - Where to start replication?
- (Changes on crashed server lost)

# Circular replication?



- Where do we start?
  - Different position on 2 and 3
  - Lag between 2 and 3
  - Lag between 3 and 4

# Circular replication

1. Create replication progress table
2. For every transaction:
  - Figure out binlog position
  - Write it to table with transaction
  - Need to use special client code
3. On failure:
  - Fetch position from replication progress table
  - Change to position and start slave

# Step 1: Replication progress

- Create replication progress table
  - **Name:** Replication\_progress
  - **Column:** Server\_id
  - **Column:** Master\_log\_file
  - **Column:** Master\_log\_pos

```
CREATE TABLE Replication_progress (  
    Server_id INT UNSIGNED,  
    Log_file CHAR(64),  
    Log_pos INT UNSIGNED,  
    PRIMARY KEY (Server_id)  
) ENGINE=MYISAM;
```

# Step 2: Transaction position

- Set AUTOCOMMIT

```
SET AUTOCOMMIT=0
```

- Lock tables needed

- This will also start the transaction

```
LOCK TABLES
```

```
  Replication_progress WRITE,  
  /* other tables */
```

- Execute transaction and commit

```
...; COMMIT;
```

# Step 2: Transaction position

- Fetch master position

```
($File, $Pos) = `SHOW MASTER STATUS`
```

- Update replication progress table

```
INSERT INTO Replication_progress  
VALUES ($Server_id, '$File', $Pos)  
ON DUPLICATE KEY  
UPDATE Log_file = '$File',  
       Log_pos = $Pos
```

- Unlock tables

```
UNLOCK TABLES
```

# Step 2: How to fail-over

- Decide fail-over server

```
$Failover_id
```

- Find position

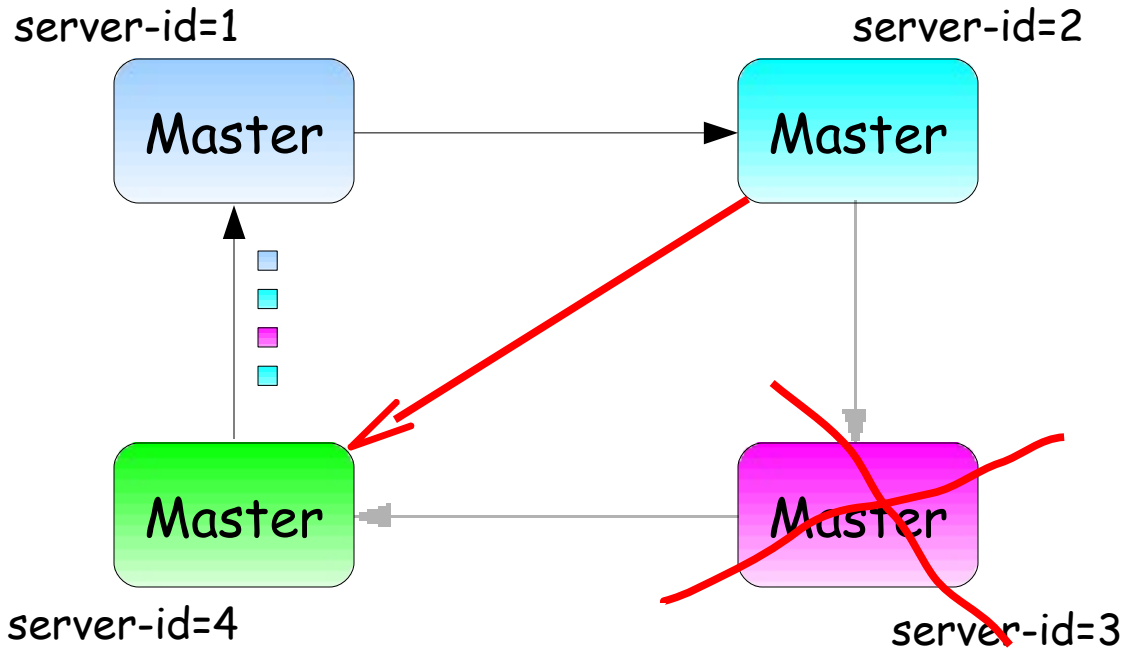
```
($File, $Pos) =  
`SELECT Log_file, Log_pos  
  FROM Replication_progress  
  WHERE Server_id = $Failover_id`
```

- Change master and start slave

```
CHANGE MASTER TO MASTER_HOST = ...,  
  MASTER_LOG_FILE = $File,  
  MASTER_LOG_POS = $Pos
```

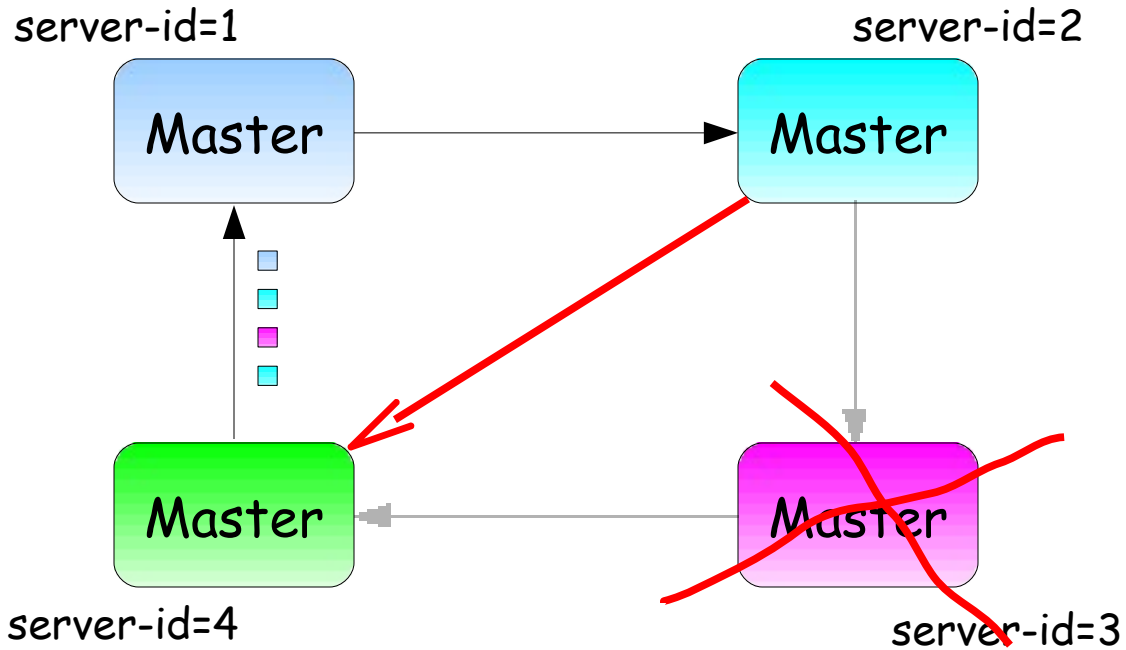
```
START SLAVE
```

# Circular replication



- What about server 3 events?
  - Leave them
  - Introduce fake server

# Circular replication



- 6.0 feature

```
CHANGE MASTER TO  
  MASTER_LOG_FILE = ...,  
  MASTER_LOG_POS = ...,  
  IGNORE_SERVER_IDS = (3);
```



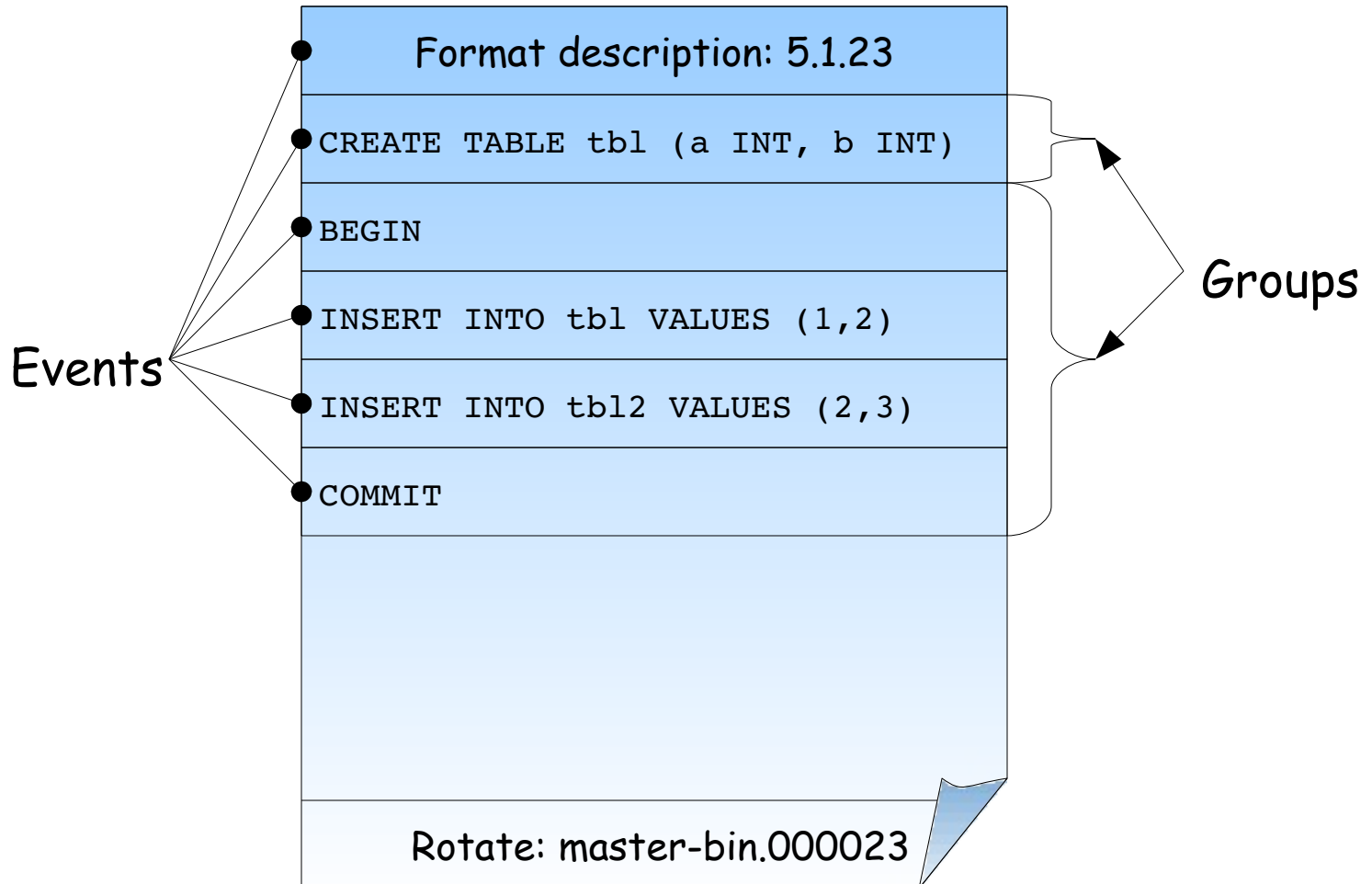
Innovation Everywhere

# The binary log

A closer look into the binary log

# Binlog events

master-bin.000022



# Statement logging

- Statements use *Query* log event
- Statements are logged *verbatim*
  - ...with some exceptions
- USE statement added
  - ... with current database

```
mysqld.1> show binlog events from 106 limit 1\G
***** 1. row *****
  Log_name: master-bin.000001
     Pos: 106
Event_type: Query
Server_id: 1
End_log_pos: 200
      Info: use `test`; CREATE TABLE tbl (a INT, b INT)
1 row in set (0.00 sec)
```

# Statement logging

- What about this statement?

```
UPDATE db1.t1, db2.t2  
SET db1.t1.a = db2.t2.a
```

- Logged with the *current* database
- Statement cannot be executed if db1 or db2 is filtered (but not both)
- Situation have to be avoided:
  - USE the right database
  - Don't qualify tables with database

# Statement logging

- Statement context events
  - User variables
  - RAND ( )
  - AUTO\_INCREMENT
- Context events written before

```
***** 1. row *****
Event_type: User var
Info: @`user`=_latin1 0x6D6174734073756E2E636F6D COLLATE latin1_swedish_ci
***** 2. row *****
Event_type: Query
Info: use `test`; INSERT INTO user VALUES (1,@user)
```

# Unsafe statements

- User-defined functions (UDFs)
  - Can do anything
- Other unsafe constructions:
  - UUID()
  - FOUND\_ROWS()
  - Two or more tables with AUTO\_INCREMENT
  - ... and more

# Statement logging

- Statements are logged:
  - *after* statement is executed
  - *before* statement is committed
- Non-transactional changes
  - Can be partially executed
  - Can cause inconsistency

# Row-based replication

- Introduced in 5.1
- Replicate actual row changes
- Can handle “difficult” statements
  - UDFs, UUID(), ...
  - Automatic switching
  - Partially executed statements
- Used for Cluster replication
- A foundation for new development

# Binlog formats

## **STATEMENT**

- Everything replicated as statement
- Same as for 5.0

## **MIXED**

- Replicates in statement format by default
- Switch to row format for unsafe statements

## **ROW**

- DML is replicated in row format
- DDL is replicated in statement format

# Using MIXED

- Server variable
  - For a single session only:  
`SET SESSION BINLOG_FORMAT=MIXED`
  - For all sessions:  
`SET GLOBAL BINLOG_FORMAT=MIXED`
- Configuration option:  
`binlog-format=mixed`

*Recommended*

# Row-based and filtering

- Individual rows are filtered
  - Filtered based on *actual* database
    - (Statement-based on *current* database)
  - Master filters on table possible
- ... but not implemented

```
UPDATE db1.t1, db2.t2  
SET db1.t1.a = db2.t2.a
```

No problems

# Row-based as a foundation

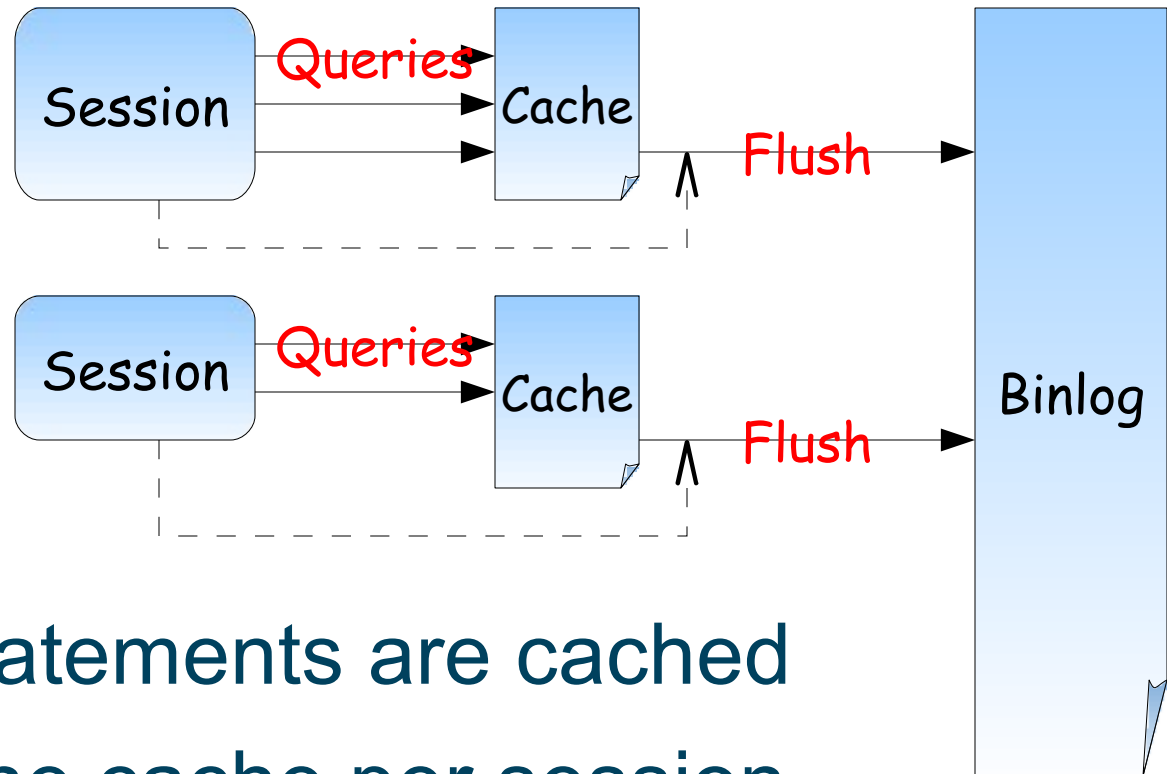
- Conflict detection and resolution *Done*
- Fine-grained filtering
  - Master filter on table
- Cluster replication *Done*
- Multi-channel replication
- Transactional behavior
  - Possibility to separate transactional and non-transactional changes in a statement
- Horizontal partitioning
  - Sending different rows to different slaves



# Statements and Transactions

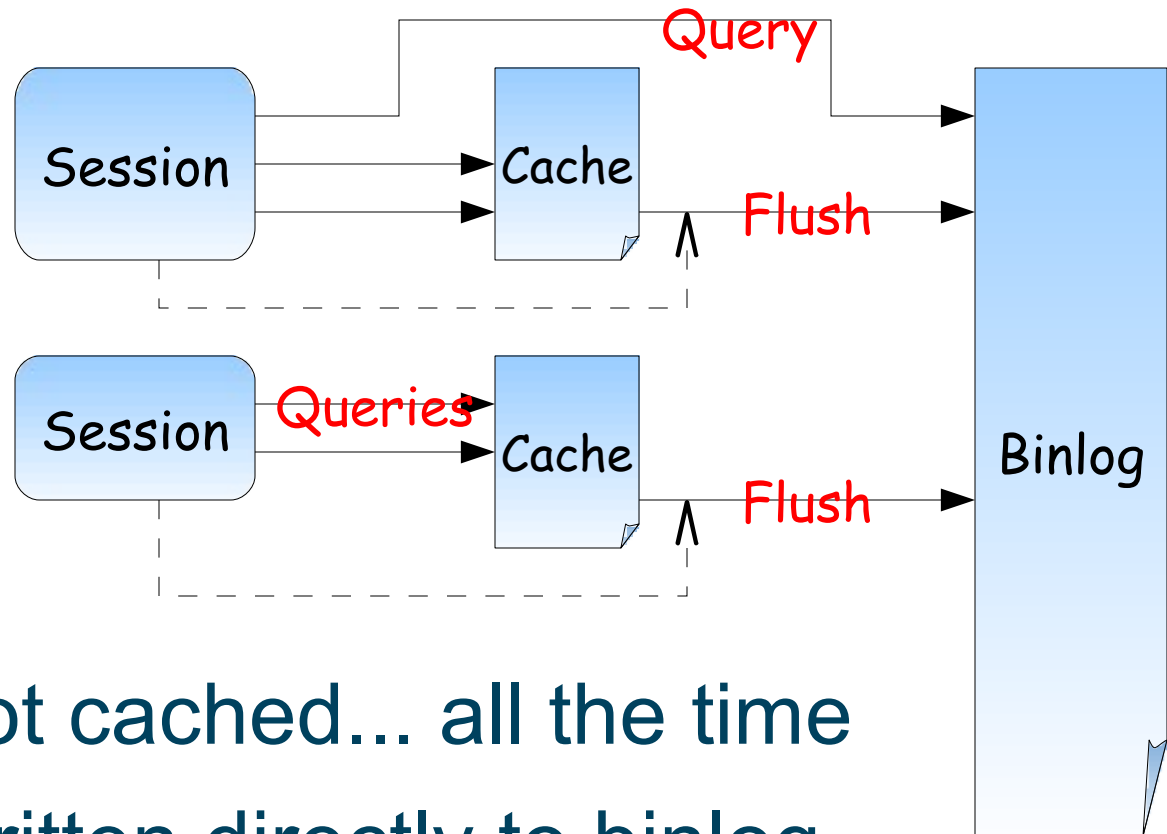
Innovation Everywhere

# Transaction cache



- Statements are cached
- One cache per session
- Cache is written to binlog on commit

# Non-transactional statements



- Not cached... all the time
- Written directly to binlog
- Locks binlog

# Non-transactional statements

- Inside a transaction
  - $<5.1.31$ :
    - If cache is not empty: **cache**
    - Otherwise: **write directly**
  - $\geq 5.1.31$ :
    - Always cached
- Outside a transaction
  - Never cached

# Non-transactional statements

```
CREATE TABLE trans (a CHAR(64)) ENGINE=INNODB;  
CREATE TABLE non_trans (a CHAR(64)) ENGINE=MYISAM;  
  
BEGIN;  
INSERT INTO trans VALUES (1),(2),(3);  
INSERT INTO non_trans SELECT * FROM trans;  
...  
COMMIT/ROLLBACK;
```

- To cache or not to cache...
- Keep safe: cache the statement

# Mixing engines in statements

```
CREATE TABLE user (  
    uid INT AUTO_INCREMENT,  
    name CHAR(64), email CHAR(64),  
    PRIMARY KEY(uid)  
) ENGINE=INNODB;
```

```
CREATE TABLE log (uid CHAR(64), comment TEXT) ENGINE=MYISAM;
```

```
CREATE TRIGGER tr_user AFTER INSERT ON user FOR EACH ROW  
    INSERT INTO log VALUES(NEW.uid, "New user added");
```

- Table `user` to track users
- Table `log` track changes to user
- Trigger `tr_user`:
  - Insert entry in `log` when user is added

# Mixing engines in statements

- Consider this statement:

```
INSERT INTO user  
VALUES (NULL, 'mats', 'mats@sun.com');
```

- Statement changes:
  - Transactional table user
  - Non-transactional table log
- Is this statement transactional?
- Shall it be written to the cache?

# Mixing engines in statements

- If treated as transactional:

```
BEGIN;  
INSERT INTO innodb_tbl VALUES...  
INSERT INTO user VALUES ...  
ROLLBACK;
```

- Master and slave inconsistent
- If treated as non-transactional:

```
BEGIN;  
INSERT INTO user VALUES ...  
ROLLBACK;
```

- Master and slave inconsistent

*Fixed in 5.1.31*

# Non-transactional statements

- Inside a transaction
  - <5.1.31:
    - If cache is not empty: **cache**
    - Otherwise: **write directly**
  - ≥5.1.31:
    - **Always cached** *This is the fix*
- Outside a transaction
  - Never cached

# Mixing engines in statements

- Don't write this:

```
BEGIN;  
INSERT INTO myisam_tbl VALUES...  
INSERT INTO innodb_tbl VALUES...  
...  
COMMIT;
```

- Write this:

```
INSERT INTO myisam_tbl VALUES...  
BEGIN;  
INSERT INTO innodb_tbl VALUES...  
...  
COMMIT;
```

# Triggers and replication

- Non-transactional trigger
  - Statement becomes non-transactional
  - Legacy from statement-based
    - 5.0: statement can be transactional
- Non-transactional “write-ahead”
  - Possible with row-based replication
  - Not implemented yet

# Events and replication

- CREATE, DROP, and ALTER
  - DDL: Replicated as statements
- **Event is disabled on slave**
  - It should **not** execute on slave
  - Executed twice otherwise
- Enabled with ALTER EVENT



# Binlog events

A closer look at the  
contents of binlog events

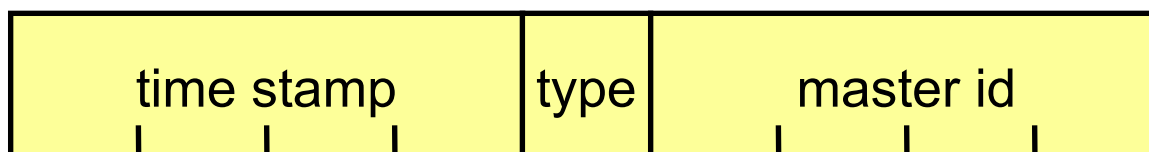
Presented by



O'REILLY

## Common Event Header – 19 bytes

Field	Length	Description
Timestamp	4 bytes	Seconds since 1970
Type	1 byte	Event type
Master Id	4 bytes	Server Id of server that created this event
Total size	4 bytes	Event total size in bytes
Master position	4 bytes	Position of next event in master binary log
Flags	2 bytes	Flags for event



# Statement-based INSERT 1/2: Query event header

```
$ mysqlbinlog --hexdump master-bin.000001
```

```
# at 235
#060420 20:16:02 server id 1 end_log_pos 351
# Position          Timestamp          Type          Master ID
# 000000eb          e2 cf 47 44      02           01 00 00 00
#      Size          Master Pos        Flags
# 74 00 00 00      5f 01 00 00      10 00
```

## Statement-based INSERT 2/2: Query event data

```
$ mysqlbinlog --hexdump master-bin.000001
```

```
# 000000fe 02 00 00 00 00 00 00 00 00
#           04 00 00 1a 00 00 00 40 |.....|
# 0000010e 00 00 ... |.....std|
# 0000011e 04 08 ... |.....test.INSE|
# 0000012e 52 54 ... |RT.INTO.t1.VALUE|
# 0000013e 53 20 ... |S...A...B...X|
# 0000014e 27 2c ... |...Y...X...X.|
# 0000015e 29 |.|
# Query      thread_id=2      exec_time=0      error_code=0
```

```
SET TIMESTAMP=1145556962;
```

```
INSERT INTO t1 VALUES ('A','B'), ('X','Y'), ('X','X');
```

# Row-based INSERT 1/2: Table map event

```
$ mysqlbinlog --hexdump master-bin.000001
```

```
# at 235
#060420 20:07:01 server id 1 end_log_pos 275
# Position          Timestamp          Type      Master ID
# 000000eb          c5 cd 47 44       13        01 00 00 00
#      Size          Master Pos        Flags
# 28 00 00 00       13 01 00 00       00 00
# 000000fe 0f 00 00 00 00 00 00 00 00
#           04 74 65 73 74 00 02 74 |.....test..t|
# 0000010e 31 00 02 fe fe           |1....|
# Table_map: `test`.`t1` mapped to number 15
BINLOG 'xc1HRBMBAAAKAAAABMBA...3QAAnQxAAL+/g==' ;
```

## Row-based INSERT 2/2: Write event

```
$ mysqlbinlog --hexdump master-bin.000001
```

```
# at 275
```

```
#060420 20:07:01 server id 1 end_log_pos 319
```

```
# Position          Timestamp          Type      Master ID
```

```
# 00000113          c5 cd 47 44       14        01 00 00 00
```

```
#      Size          Master Pos        Flags
```

```
# 2c 00 00 00       3f 01 00 00       10 00
```

```
# 00000126 0f 00 00 00 00 00 01 00
```

```
#      02 ff f9 01 41 01 42 f9 | .....A.B. |
```

```
# 00000136 01 58 01 59 f9 01 58 01
```

```
#      58                          | .X.Y..X.X |
```

```
# Write_rows: table id 15
```

```
BINLOG 'xc1HRBQBAAAALAAAAD...EBQvkBwAFZ+QFYAVg=' ;
```



# Cluster replication

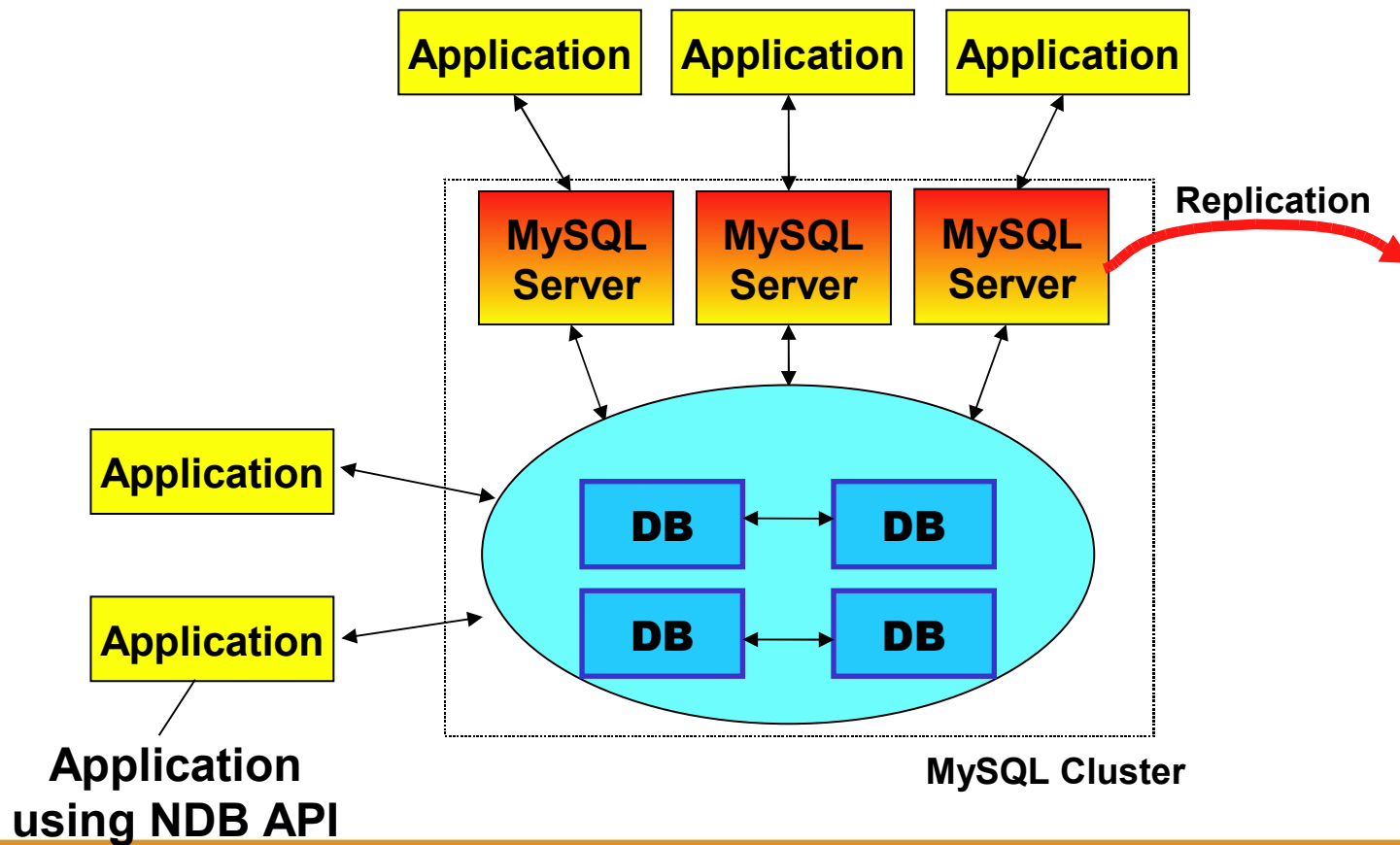
Presented by



O'REILLY

# MySQL Cluster Replication

## Where to get the log events?



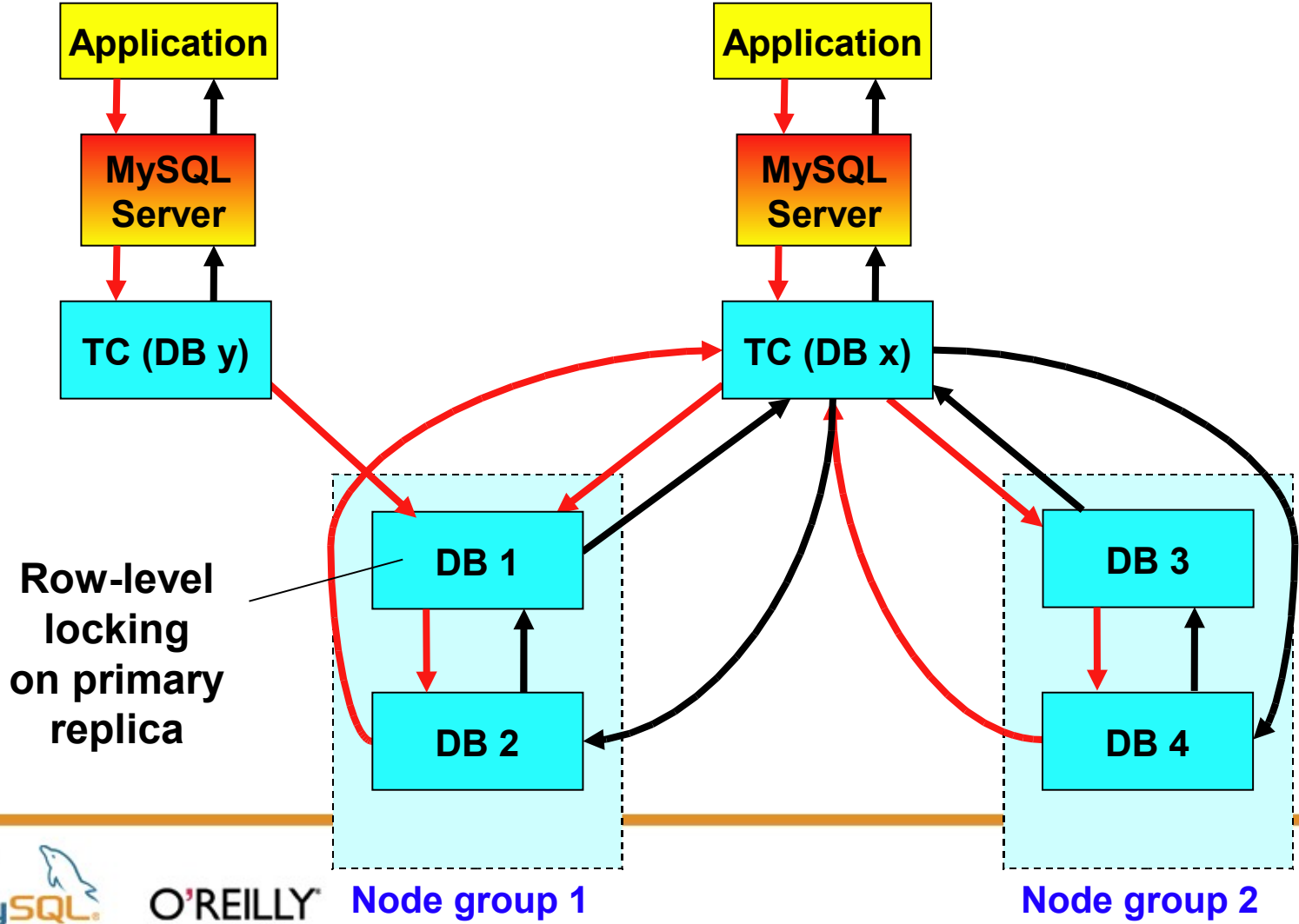
Presented by



O'REILLY

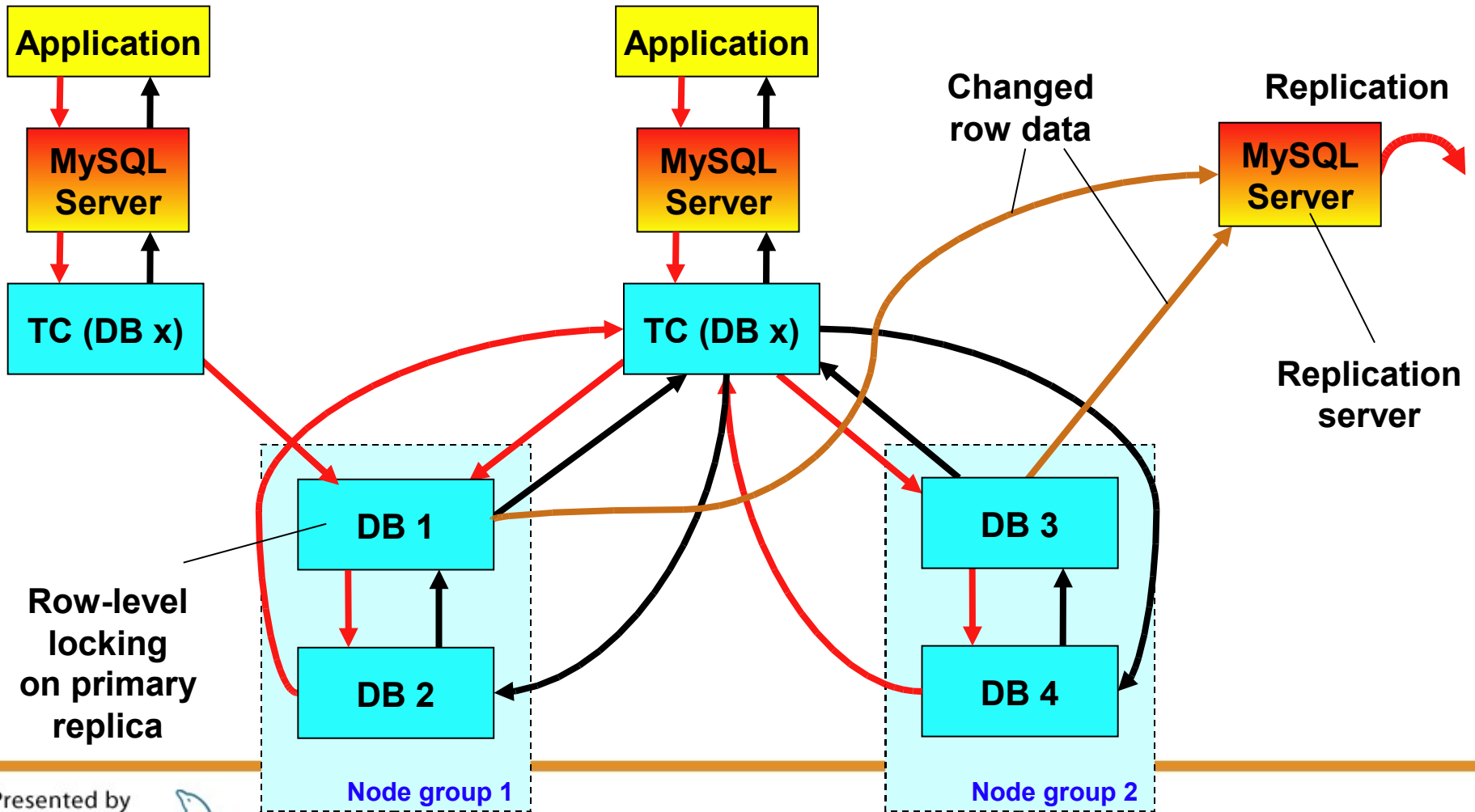
# MySQL Cluster Replication

## Concurrency control inside master cluster



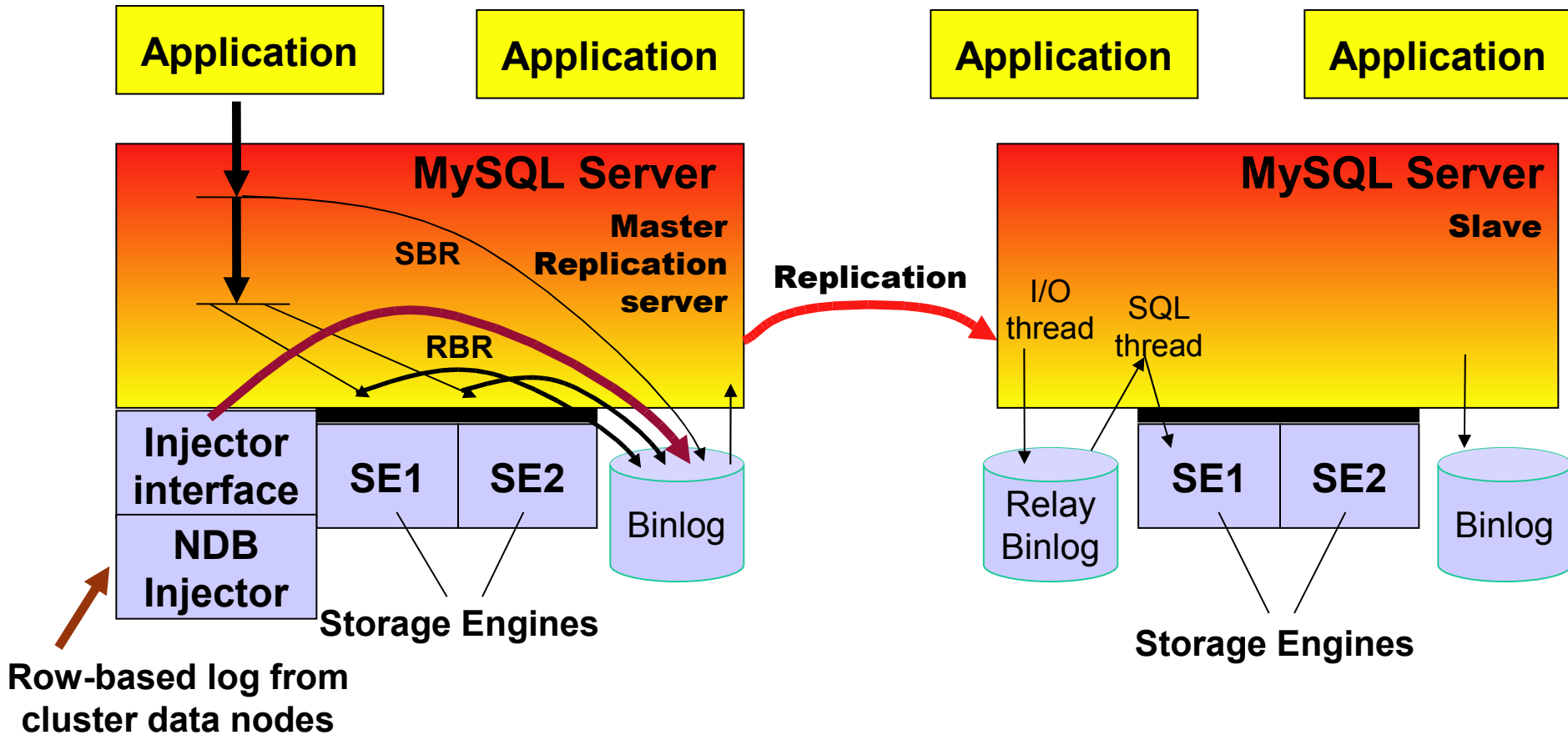
# MySQL Cluster Replication

## Log shipping inside master cluster



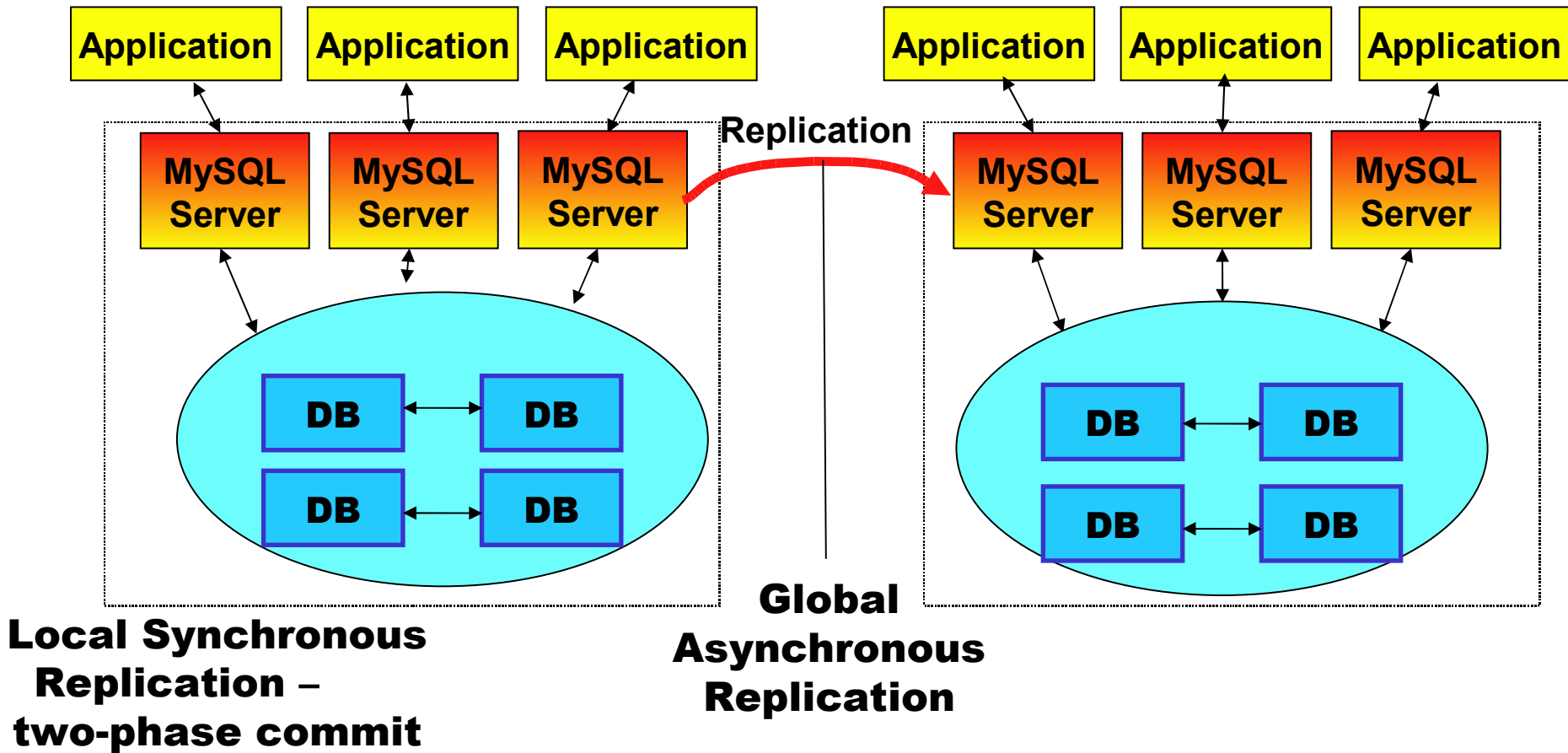
# MySQL Replication Architecture

## MySQL 5.1



# MySQL Cluster Replication

## Behaves like ordinary MySQL Replication





# The End

Mats Kindahl  
[mats@sun.com](mailto:mats@sun.com)

Lars Thalmann  
[lars.thalmann@sun.com](mailto:lars.thalmann@sun.com)

Presented by



O'REILLY