DeNA

Automated, Non-Stop MySQL Operations and Failover

Yoshinori Matsunobu

Principal Infrastructure Architect, DeNA

Former APAC Lead MySQL Consultant at MySQL/Sun/Oracle

Yoshinori.Matsunobu@gmail.com

http://yoshinorimatsunobu.blogspot.com/

Table of contents

- Automating master failover (main topic)
- Minimizing downtime at master maintenance

Company Introduction: DeNA and Mobage

- One of the largest social game providers in Japan
  - Both social game platform and social games themselves
  - Subsidiary ngmoco:) in SF
- Japan localized phone, Smart Phone, and PC games
- 2-3 billion page views per day
- 25+ million users
- 700+ MySQL servers
- 1.3B$ revenue in 2010

HA Requirements for social games

- Requirements about high availability and integrity are quite high
  - Paid service dramatically raises expectations from users
    - "I haven't received a virtual item I paid for"
    - "My HP/MP fell after I used non-free recovery item"
  - Long downtime causes huge negative impacts on revenue
  - Planned maintenance is not impossible, if properly planned and announced
    - Traffic at 5 am is less than 1/5 compared to 11 pm
    - Much better than unplanned downtime

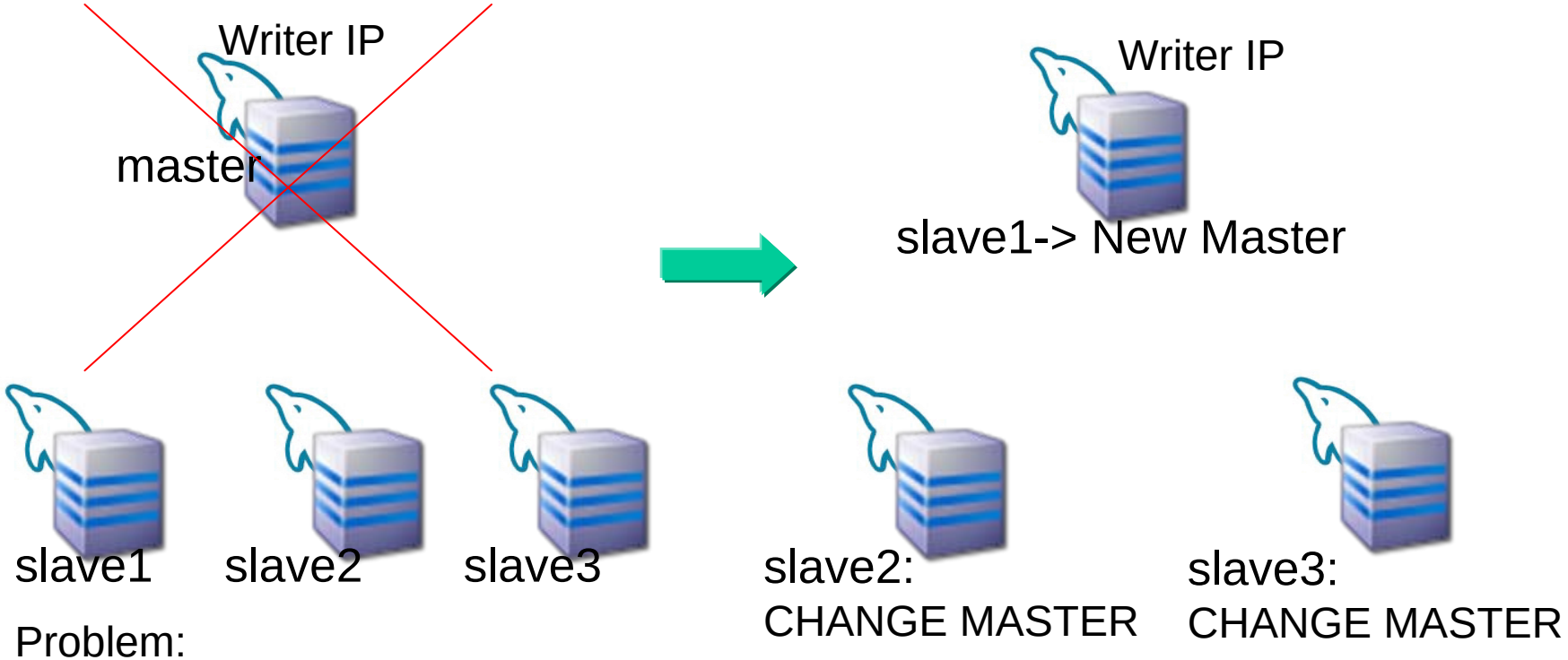
The goal is “No Single Point of Failure”

DeNA

- We operate 700+ MySQL servers at DeNA
  - More than 150 {master, slaves} pairs
  - Mainly MySQL 5.0 and 5.1
- Statistically MySQL master went down once per a few months
  - In many times caused by hangs on Linux or H/W failures
  - Manual failover should be avoided if possible, to minimize downtime
- It is easy to make slaves not single point of failure
  - Just running two or more slaves
- It is not trivial to make masters not single point of failure

- We want to automate master failover and slave promotion
  - On regular MySQL 5.0/5.1, and 5.5+
    - We don’t want to spend time for significant architecture changes on legacy running services
  - Without losing performance significantly
  - Without spending too much money

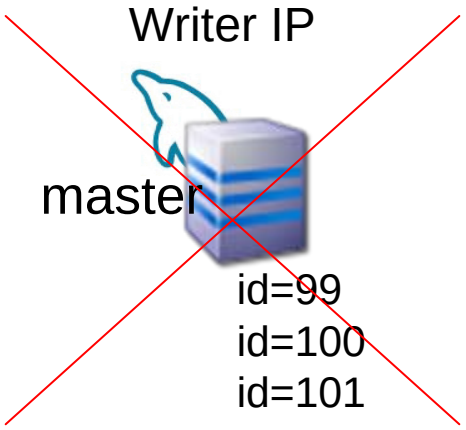
Master Failover: What is the problem?



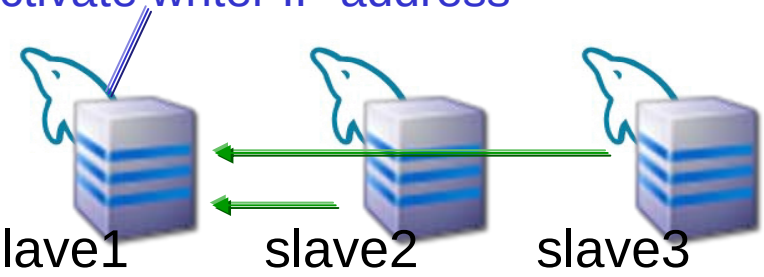
Problem:
When a master goes down, the system also goes down until *manual* master failover completes (you can't do writes). It is not uncommon to take one hour or even more to recover.

Objective:
Automate master failover. That is, pick one of the appropriate slaves as a new master, making applications send write traffics to the new master,
then starting replication again.

Failure Example (1)



Get current binlog position (file1,pos1)
Grant write access
Activate writer IP address



slave1	slave2	slave3
id=99	id=99	id=99
id=100	id=100	id=100
id=101	id=101	id=101

All slaves have received all binlog events from the crashed master.

Any slave can be a new master, without recovering any data

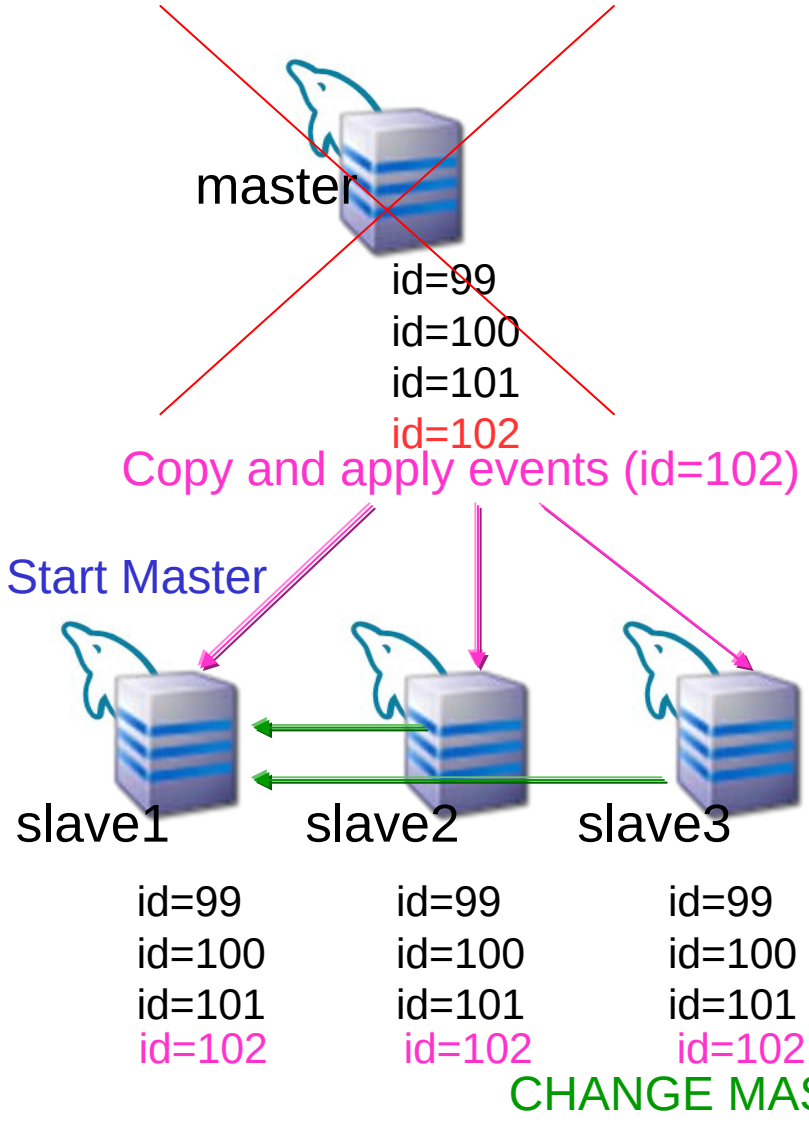
Example: picking slave 1 as a new master

Slave 2 and 3 should execute
CHANGE MASTER MASTER_HOST= ‘slave1’ …;
START SLAVE;

This is the easiest scenario.
But not all times it is so lucky.

Execute CHANGE MASTER TO MASTER_HOST=‘slave1’, MASTER_LOG_FILE=‘file1’, MASTER_LOG_POS=pos1;

Failure Example (2)



All slaves have received same binlog events from the crashed master.

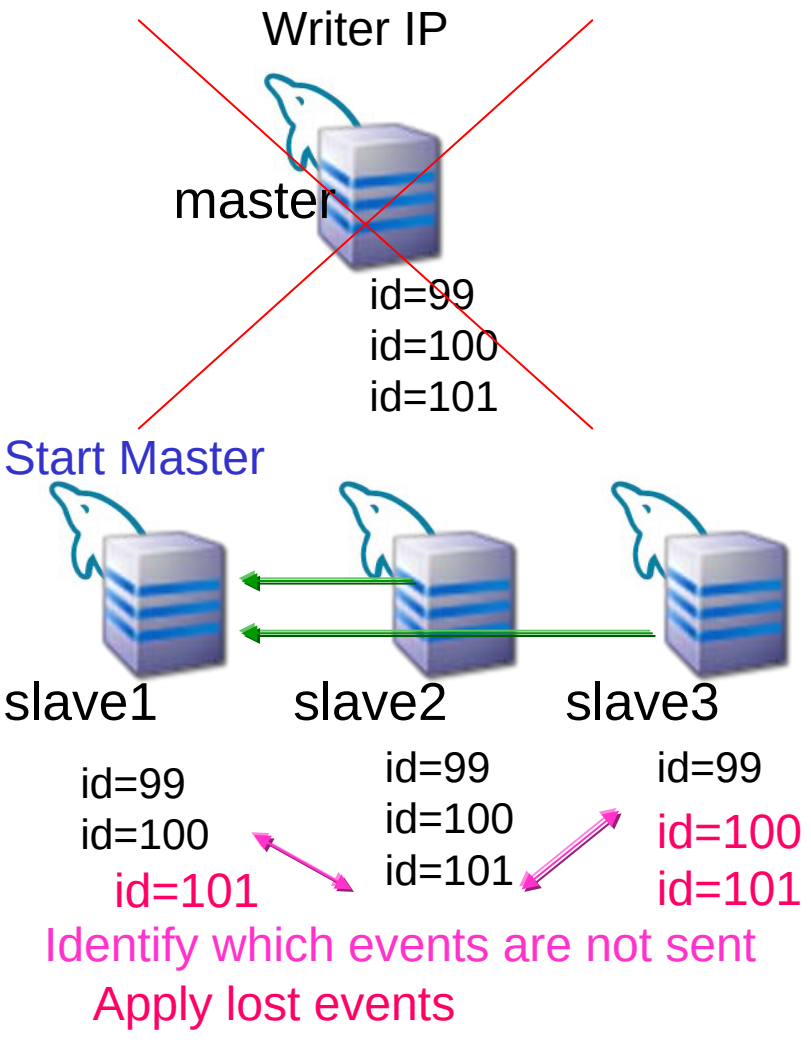
But the crashed master has some events that have not been sent to slaves yet.

id=102 will be lost if you promote one of slaves to a new master.

If the crashed master is reachable (via SSH) and binlog file is readable, you should save binlog (id=102) before promoting a slave to a new master.

Using Semi-Synchronous replication greatly reduces the risk of this scenario.

Failure Example (3)

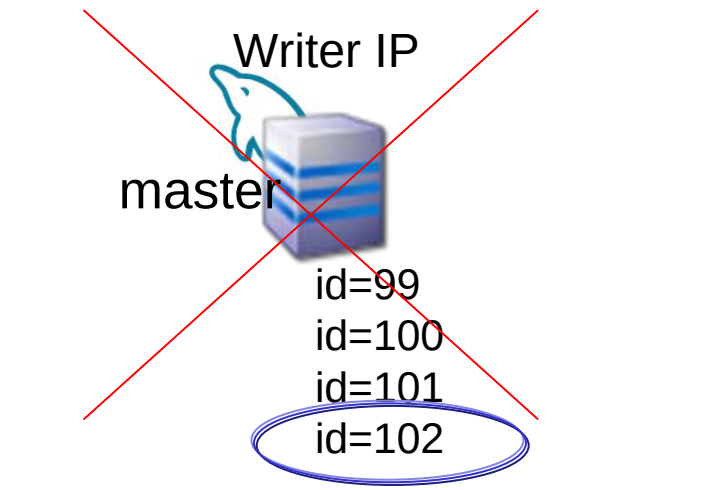


Some slaves have events which other slaves have not received yet.

You need to pick events from the latest slave (slave 2), and apply to other slaves so that all slaves will be consistent.
(Sending id=101 to slave 1, sending id=100 and 101 to slave 3)

The issues are:
- How can we identify which binlog events are not sent?
- How can we make all slaves eventually consistent?

Master Failover: What makes it difficult?



MySQL replication is asynchronous.

It is likely that some (or none of) slaves have not received all binary log events from the crashed master.

It is also likely that only some slaves have received the latest events.

In the left example, id=102 is not replicated to any slave.

slave 2 is the latest between slaves, but slave 1 and slave 3 have lost some events.
It is necessary to do the following:
- Copy id=102 from master (if possible)
- Apply all differential events, otherwise data inconsistency happens.

Current HA solutions and problems

DeNA

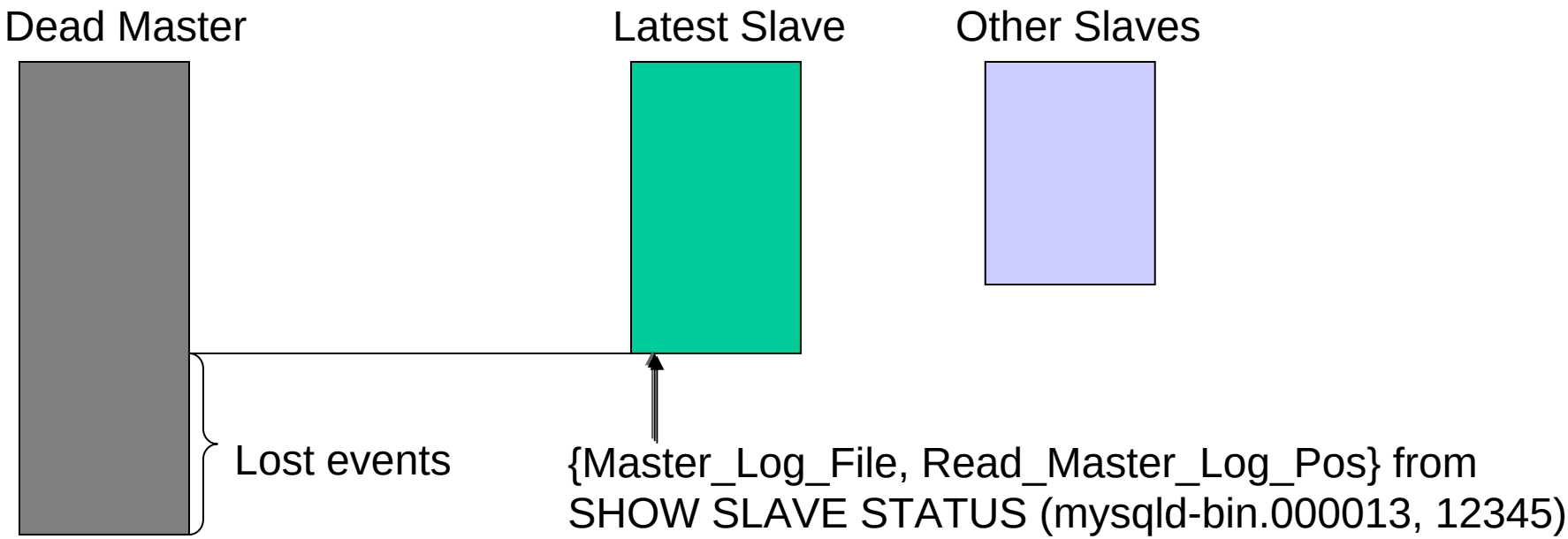
- Heartbeat + DRBD
  - Cost: Additional passive master server (not handing any application traffic) is needed
  - Performance: To make HA really work on DRBD replication environments, innodb-flush-log-at-trx-commit and sync-binlog must be 1. But these kill write performance
  - Otherwise necessary binlog events might be lost on the master. Then slaves can't continue replication, and data consistency issues happen
- MySQL Cluster
  - MySQL Cluster is really Highly Available, but unfortunately we use InnoDB
- Semi-Synchronous Replication (5.5+)
  - Semi-Sync replication greatly minimizes the risk of "binlog events exist only on the crashed master" problem
  - It guarantees that *at least one* (not all) slaves receive binlog events at commit. Some of slaves might not receive all binlog events at commit.
- Global Transaction ID
  - On mysql side, it's not supported yet. Adding global transaction Id within binary logs require binlog format change, which can't be done in 5.1/5.5.
    - Check Google's Global Transaction ID patch if you're interested
  - There are ways to implement global tx ID on application side, but it's not possible without accepting complexity, performance, data loss, and/or consistency problems

More concrete objective

DeNA

- Make master failover and slave promotion work
  - Saving binary log events from the crashed master (if possible)
    - Semi-synchronous replication helps too
  - Identifying the latest slave
  - Applying differential relay log events to other slaves
  - Applying saved binary log events from master
  - Promoting one of the slaves to a new master
  - Making other slaves replicate from the new master
- Automate the above procedure
  - Master failure should also be detected automatically
- Do the above
  - Without introducing too much complexity on application side
  - With 5.0/5.1 InnoDB
  - Without losing performance significantly
  - Without spending too much money

Saving binlog events from (crashed) master



mysqlbinlog --start-position=12345 mysqld-bin.000013 mysqld-bin.000014….

- If the dead master is reachable via SSH, and binary logs are accessible (Not H/W failure, i.e. InnoDB data file corruption on the master), binlog events can be saved.
- Lost events can be identified by checking {Master_Log_File, Read_Master_Log_Pos} on the latest slave + mysqlbinlog
- Using Semi-Synchronous replication greatly reduces the risk of events loss

Understanding SHOW SLAVE STATUS

```
mysql> show slave status¥G
       Slave_IO_State: Waiting for master to send event
          Master_Host: master_host
          Master_User: repl
          Master_Port: 3306
        Connect_Retry: 60
      Master_Log_File: mysqld-bin.000980
  Read_Master_Log_Pos: 629290122
       Relay_Log_File: mysqld-relay-bin.000005
        Relay_Log_Pos: 26087338
Relay_Master_Log_File: mysqld-bin.000980
     Slave_IO_Running: Yes
    Slave_SQL_Running: Yes
      Replicate_Do_DB: db1
...
          Last_Errno: 0
          Last_Error:
   Exec_Master_Log_Pos: 629290122
Seconds_Behind_Master: 0
        Last_IO_Errno: 0
        Last_IO_Error:
       Last_SQL_Errno: 0
```

- {Master_Log_File, Read_Master_Log_Pos} :

  The position in the current master binary log file up to which the I/O thread has read.

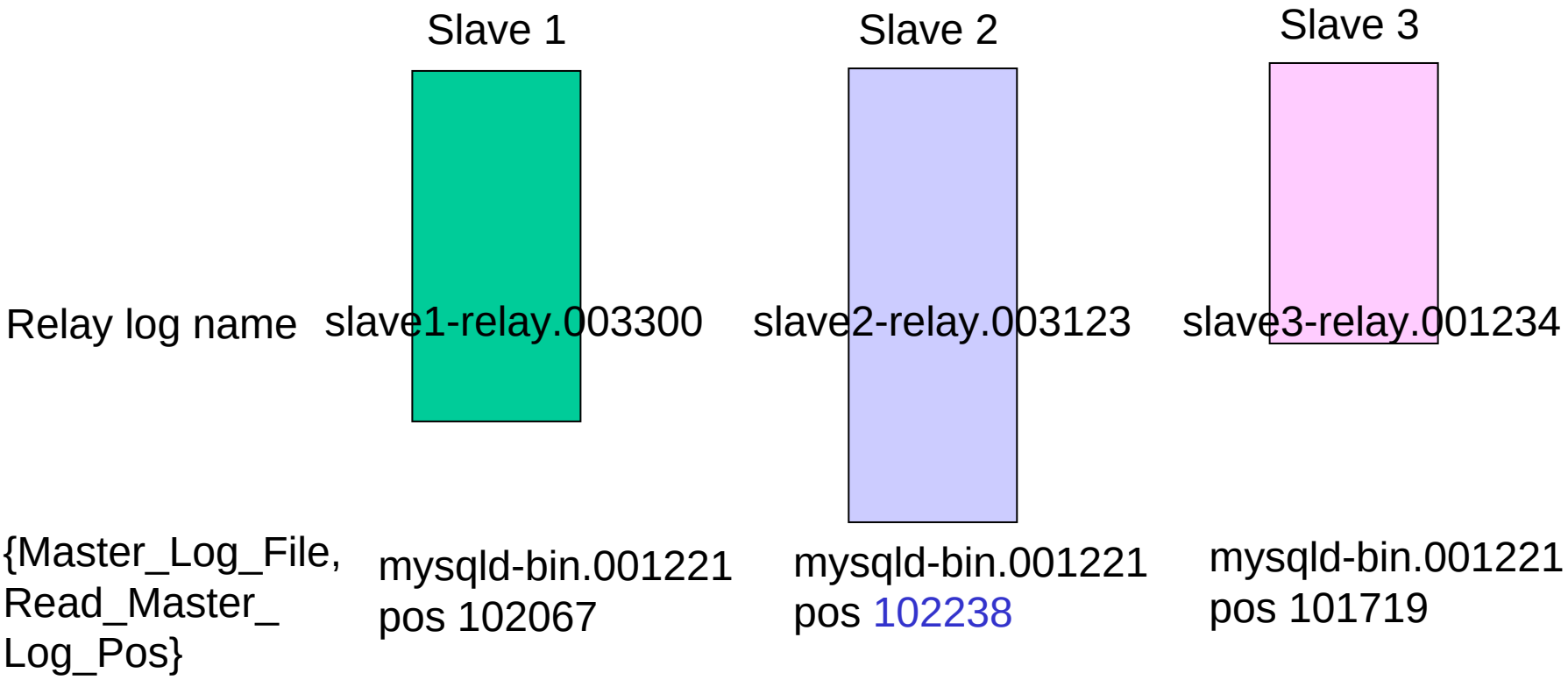
- {Relay_Master_Log_File, Exec_Master_Log_Pos} :

  The position in the current master binary log file up to which the SQL thread has read and executed.

- {Relay_Log_File, Relay_Log_Pos} :

  The position in the current relay log file up to which the SQL thread has read and executed.

Identifying the latest slave



- Relay log name/position is not helpful to identify the latest slave, because relay log name/position is independent from slaves
- By comparing {Master_Log_File, Read_Master_Log_Pos}, you can identify the latest slave
  - Slave 2 is the latest

Next issue: Applying diffs to other slaves

- How can we identify which binlog events need to be applied to each slave?

Identifying what events need to be applied

- Since we know all slave's master position, by comparing these positions, generating differential relay log events should be possible
- There is no simple way to generate differential relay log events based on master's log file/position

Relay log internals: “at” and “end_log_pos”

```
[user@slave2] mysqlbinlog slave2-relay-bin.003123
# at 106
#101210  4:19:03 server id 1384  end_log_pos 0
Rotate to mysqld-bin.001221  pos: 4
…
# at 101835
#110207 15:43:42 server id 1384  end_log_pos 101764
Query   thread_id=1784    exec_time=0     error_code=0
SET TIMESTAMP=1297061022/*!*/;
BEGIN /*!*/;
# at 101910
#110207 15:43:42 server id 1384  end_log_pos 102067
Query   thread_id=1784    exec_time=0     error_code=0
SET TIMESTAMP=1297061022/*!*/;
update …….  /*!*/;
# at 102213
#110207 15:43:42 server id 1384  end_log_pos 102211
Query   thread_id=1784    exec_time=0     error_code=0
SET TIMESTAMP=1297061022/*!*/;
update …….  /*!*/;
# at 102357
#110207 15:43:42 server id 1384  end_log_pos 102238
Xid = 12951490691
COMMIT/*!*/;
EOF
```

- “# at xxx” corresponds to relay log position of the slave. This is not master’s binlog position. Each slave might have different relay log position for the same binary log event.
- end_log_pos corresponds to the master’s binary log position. This is unique between slaves.
- At the beginning of the relay log file, normally master’s binary log file name is written.
- end_log_pos of the tail of the last relay log should be equal to {Master_Log_File, Read_Master_Log_Pos} from SHOW SLAVE STATUS.

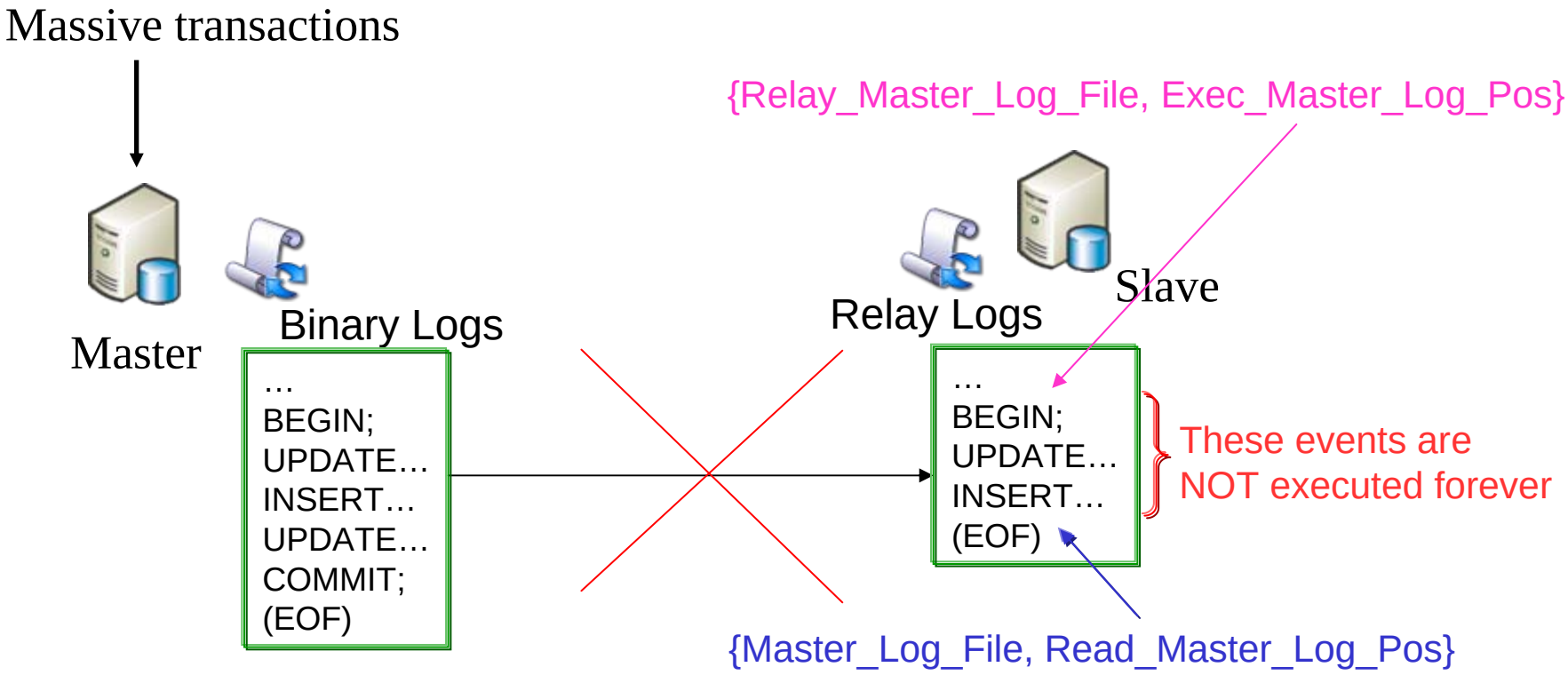
Relay log internals: How to identify diffs

```
[user@slave2] mysqlbinlog slave2-relay-bin.003123
…
# at 101807
#110207 15:43:42 server id 1384  end_log_pos 101719
Xid = 12951490655
COMMIT/*!*/;
# at 101835
#110207 15:43:42 server id 1384  end_log_pos 101764
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
BEGIN /*!*/;
# at 101910
#110207 15:43:42 server id 1384  end_log_pos 102067
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
update …….  /*!*/;
# at 102213
#110207 15:43:42 server id 1384  end_log_pos 102211
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
update …….  /*!*/;
# at 102357
#110207 15:43:42 server id 1384  end_log_pos 102238
Xid = 12951490691
COMMIT/*!*/;
EOF
```

```
[user@slave3] mysqlbinlog slave3-relay-bin.001234
…
# at 234567
#110207 15:43:42 server id 1384  end_log_pos 101719
Xid = 12951490655
COMMIT/*!*/;
EOF
```

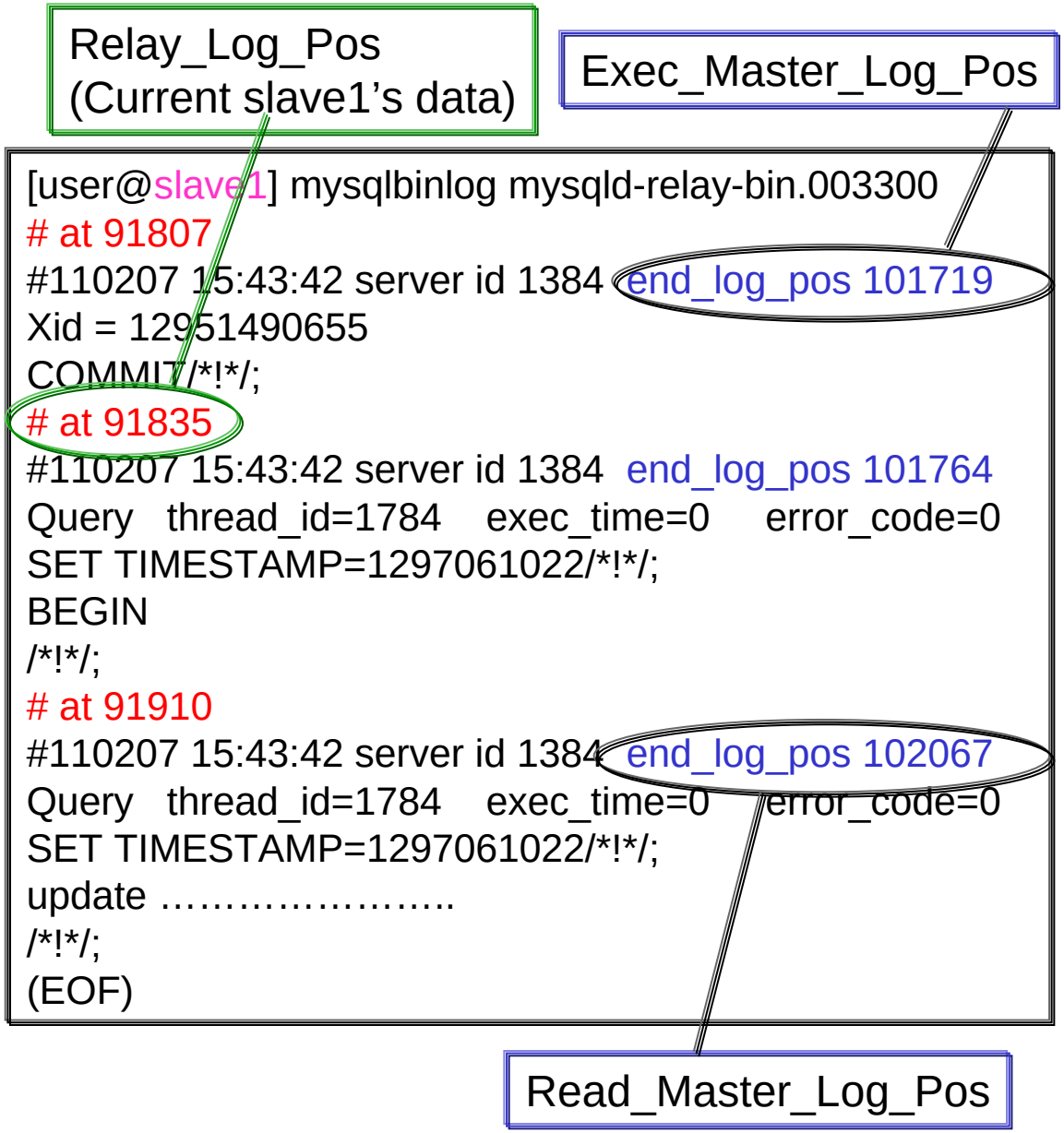
- Slave 2 has received more binlog events than Slave 3
- Check the last end_log_pos on the behind slave (101719 at Slave 3)
- Search Slave 2's relay log where end_log_pos == 101719
- Events from relay log position 101835 are lost on slave 3
- mysqlbinlog --start-position=101835 should be applied to slave 3

Relay log and “Partial Transaction”



- Alive slave IO thread writes valid relay log events, so invalid (can’t read) events should not be written to the relay log
- But if master crashes while sending binary logs, it is likely that only some parts of the events are sent and written on slaves.
- In this case, slave does not execute the last (incomplete) transaction.
  - {Master_Log_File, Read_Master_Log_Pos} points to the end of the relay log, but {Relay_Master_Log_File, Exec_Master_Log_Pos} will point to the last transaction commit.

Lost transactions



- In some unusual cases, relay logs are not ended with transaction commits
  - i.e. running very long transactions
- Read_Master_Log_Pos always points to the end of the relay log's end_log_pos
- Exec_Master_Log_Pos points to the end of the transaction's end_log_pos (COMMIT)
- In the left case, Exec_Master_Log_Pos == Read_Master_Log_Pos is never true
- Slave 1's SQL thread will never execute BEGIN and UPDATE statements
- Unapplied events can be generated by mysqlbinlog –start-position=91835

Recovering lost transactions

DeNA

```
[user@slave2] mysqlbinlog mysqld-relay-bin.003123
# at 106
#101210  4:19:03 server id 1384  end_log_pos 0
Rotate to mysqld-bin.001221  pos: 4
…
# at 101807
#110207 15:43:42 server id 1384  end_log_pos 101719
Xid = 12951490655
COMMIT/*!*/;
# at 101835
#110207 15:43:42 server id 1384  end_log_pos 101764
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
BEGIN
/*!*/;
# at 101910
#110207 15:43:42 server id 1384  end_log_pos 102067
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
update 1……..
/*!*/;
# at 102213
#110207 15:43:42 server id 1384  end_log_pos 102211
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
update 2……..
/*!*/;
# at 102357
#110207 15:43:42 server id 1384  end_log_pos 102238
Xid = 12951490691
COMMIT/*!*/; (EOF)
```

(B)

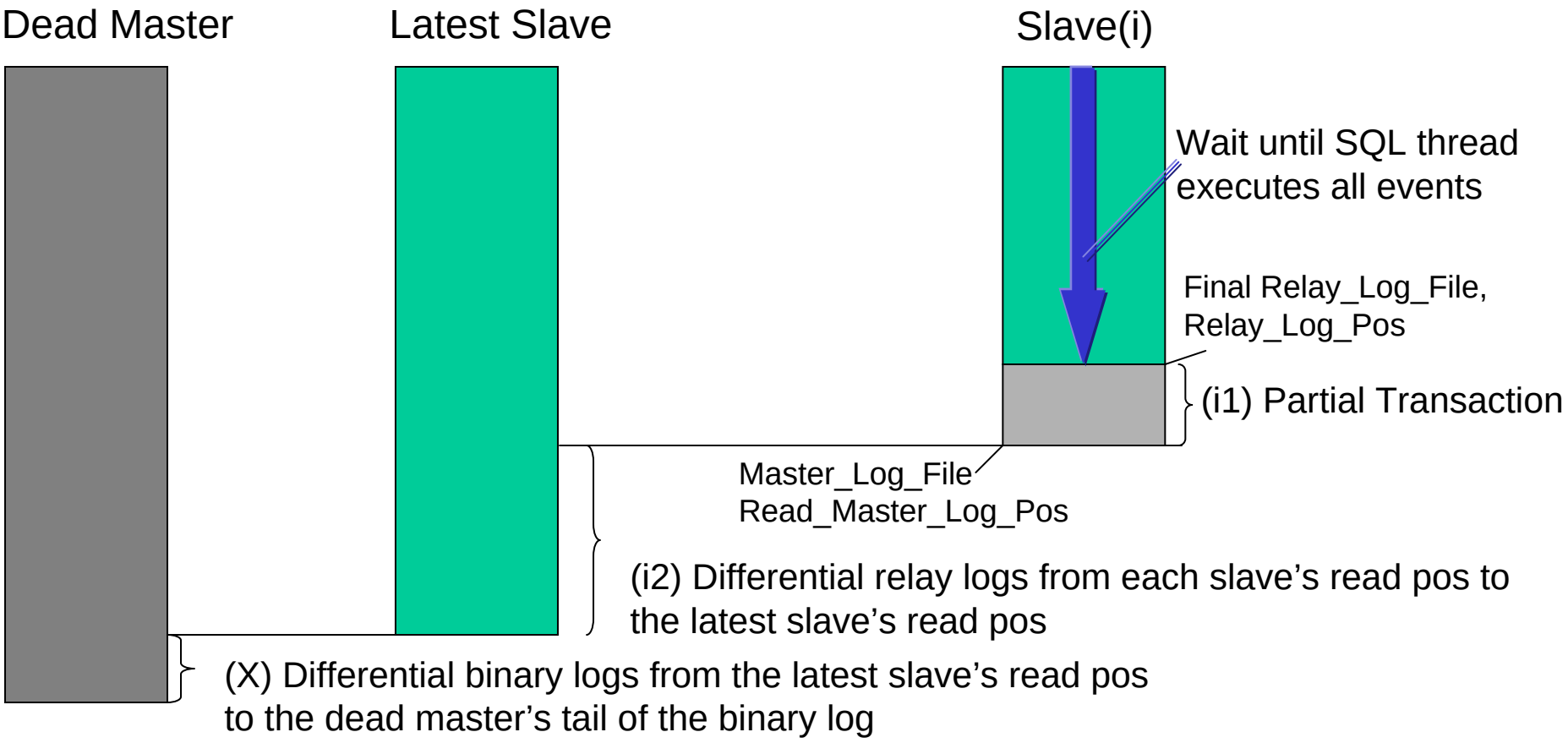
```
[user@slave1] mysqlbinlog mysqld-relay-bin.003300
# at 106
#101210  4:19:03 server id 1384  end_log_pos 0
Rotate to mysqld-bin.001221  pos: 4
…
# at 91807
#110207 15:43:42 server id 1384  end_log_pos 101719
Xid = 12951490655
COMMIT/*!*/;
# at 91835
#110207 15:43:42 server id 1384  end_log_pos 101764
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
BEGIN
/*!*/;
# at 91910
#110207 15:43:42 server id 1384  end_log_pos 102067
Query   thread_id=1784   exec_time=0   error_code=0
SET TIMESTAMP=1297061022/*!*/;
update 1……..
/*!*/;  (EOF)
```

(A)

Relay_Log_Pos
(current slave1's pos)

- The second update event is lost on slave 1, which can be sent from slave 2
- The first update event is not executed on slave 1's SQL thread
- (A) + (B) should be applied on slave 1, wichin the same transaction

Steps for recovery



- On slave(i),
  - Wait until the SQL thread executes events
  - Apply i1 -> i2 -> X
    - On the latest slave, i2 is empty

Design notes

- Trimming ROLLBACK events from mysqlbinlog
- Purging relay logs
- Identifying whether SQL thread has really executed all events
- Handling malicious queries
- Parallel recovery on multiple slaves
- Row based format

mysqlbinlog and ROLLBACK events

```
[user@slave1] mysqlbinlog slave1-relay.003300 --position=91835
# at 91835
#110207 15:43:42 server id 1384  end_log_pos 101764
Query   thread_id=1784    exec_time=0     error_code=0
SET TIMESTAMP=1297061022;
BEGIN
# at 91910
#110207 15:43:42 server id 1384  end_log_pos 102067
Query   thread_id=1784    exec_time=0     error_code=0
SET TIMESTAMP=1297061022;
update ……..
ROLLBACK; /* added by mysqlbinlog */
```

```
[user@slave2] mysqlbinlog slave2-relay.003123
# at 4
#101221 20:48:00 server id 1071  end_log_pos 107        Start: binlog
v 4, server v 5.5.8-log created 101221 20:48:00
ROLLBACK;
BINLOG '
8JMQTQ8vBAAAZwAAAGsAAAAAAQANS41LjgtbG9nAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAEzgNAAgAEgAEBAQEEgAAVAA
EGggAAAAICAgCAA== '/*!*/;
# at 102213
#110207 15:43:42 server id 1384  end_log_pos 102211      …
SET TIMESTAMP=1297061022/*!*/;
update ……..
# at 102357
#110207 15:43:42 server id 1384  end_log_pos 102238
Xid = 12951490691
COMMIT/*!*/;
ROLLBACK; /* added by mysqlbinlog */
```

- mysqlbinlog adds a ROLLBACK statement at the end of the generated file
- mysqlbinlog may add a ROLLBACK statement and/or an equivalent BINLOG event at the beginning of the generated file (included in the START event)
- If ROLLBACK is executed in the middle of the transaction, database will be inconsistent
- Trimming these ROLLBACK queries/events from mysqlbinlog outputs is needed
- Do not trim necessary rollback statements (i.e. BEGIN; updating non-trans table, updating trans table, ROLLBACK)

Purging relay logs

- By default, when SQL thread has read and executed the whole relay log file, SQL thread automatically removes it.
  - Because it is not needed by the SQL thread anymore
  - But for recovering other slaves, the old relay logs might be needed
- SET GLOBAL relay_log_purge=0, and set it in my.cnf
- Side effect:
  - Relay log files will sooner or later occupy the whole disk space
    - No similar features like expire_logs_days for binary logs
    - Schedule the following batch job will help
      * SET GLOBAL relay_log_purge=1;
      * FLUSH LOGS;
      * Waiting for a while so that SQL thread switches the log file (old logs are removed)
      * SET GLOBAL relay_log_purge=0;
  - When SQL thread reaches the end of the relay log file and if relay_log_purge equals to 1, the SQL thread removes all of the relay logs it has executed so far
    - No way to remove “all relay logs before yesterday”
    - Invoking cron jobs at the same time on the all slaves will cause “no relay log found for recovery” situation

Tips: Removing lots of large files

- Another serious side effect:
  - SQL thread removes all relay log files when it reaches the end of the relay log
  - When you set relay_log_purge=1 per day, the total relay log file size might reach 10GB or (much) more
  - Dropping lots of large files take very long time on ext3
  - SQL thread stops until removing all relay logs
    - Might take 90 seconds to drop 30*1GB files
- Solution: Creating hard links
  - foreach (relay_logs)
    - ln /path/to/relay_log /path/to/archive_dir/
  - SET GLOBAL relay_log_purge=1; FLUSH LOGS; SET GLOBAL relay_log_purge=0;
  - rm –f /path/to/archive_dir/*

How to identify whether SQL thread has executed all events

DeNA

- You need wait until SQL thread has executed all events
- SELECT MASTER_POS_WAIT (<Master_Log_File>,<Read_Master_Log_Pos>) may not work
  - MASTER_POS_WAIT() blocks until the slave has read and applied all updates up to the specified position in the master log.
  - If only part of the transactions are sent to the slave, SQL thread will never execute up to Read_Master_Log_Pos.

- Check SHOW PROCESSLIST outputs
  - If there is a thread of “system user” that has “^Has read all relay log; waiting for the slave I/O thread to update it” state, the SQL thread has executed all events.

```
mysql> show processlist¥G
     Id: 14
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 5769
  State: Has read all relay log; waiting for the slave I/O thread
to update it
   Info: NULL
```

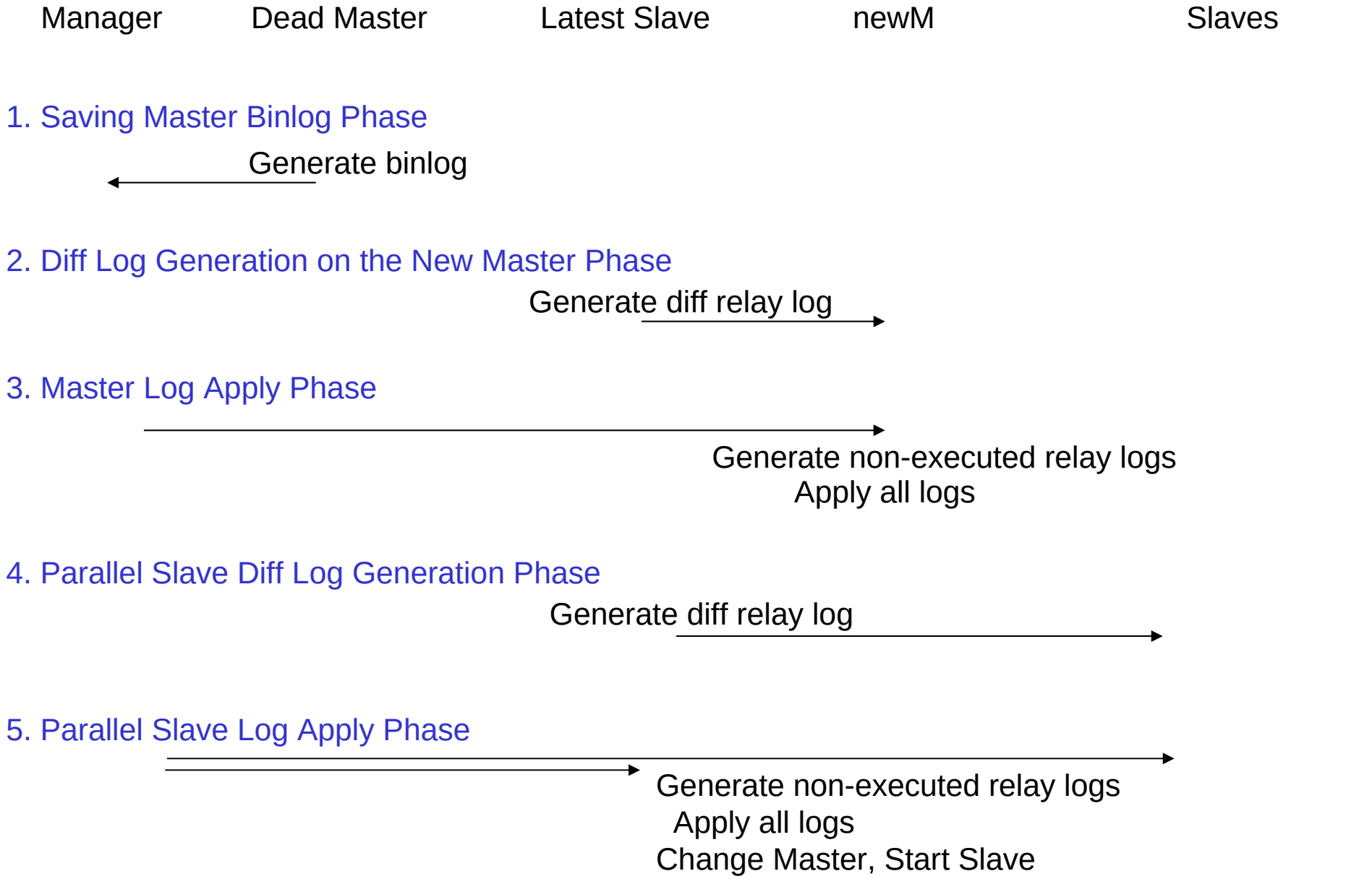
Malicious queries

- Some malicious queries might cause recovery problems
  - insert into t1 values(0,0,"ROLLBACK);
    # at 15465
    #110204 17:02:33 server id 1306 end_log_pos 1662 Query thread_id=30069 exec_time=0 error_code=0
    ROLLBACK");
  - Problems happen if end_log_pos value matches the target position
  - Use mysqlbinlog --base64-output=always to identify starting position
    - Query events are converted to row format. Base64 row format never contains malicious strings
    - Supported in mysqlbinlog from MySQL 5.1 or higher, but can work with MySQL 5.0 server, too
    - After identifying starting relay log position, generate events by normal mysqlbinlog arguments (printing query events don't cause problems here)
- 5.1 mysqlbinlog can read 5.0/5.1 binlog format
  - Use --base64-output=never for 5.0 mysqld to suppress printing BINLOG events

Parallel Recovery

- In some cases many (10 or more) slaves are deployed
- Each slave can be recovered in parallel
- Relay logs are deleted once the slave executes CHANGE MASTER.
- You must not execute CHANGE MASTER on the latest slave until you generate diff relay logs for all the rest slaves

Recovery procedure



Row based format

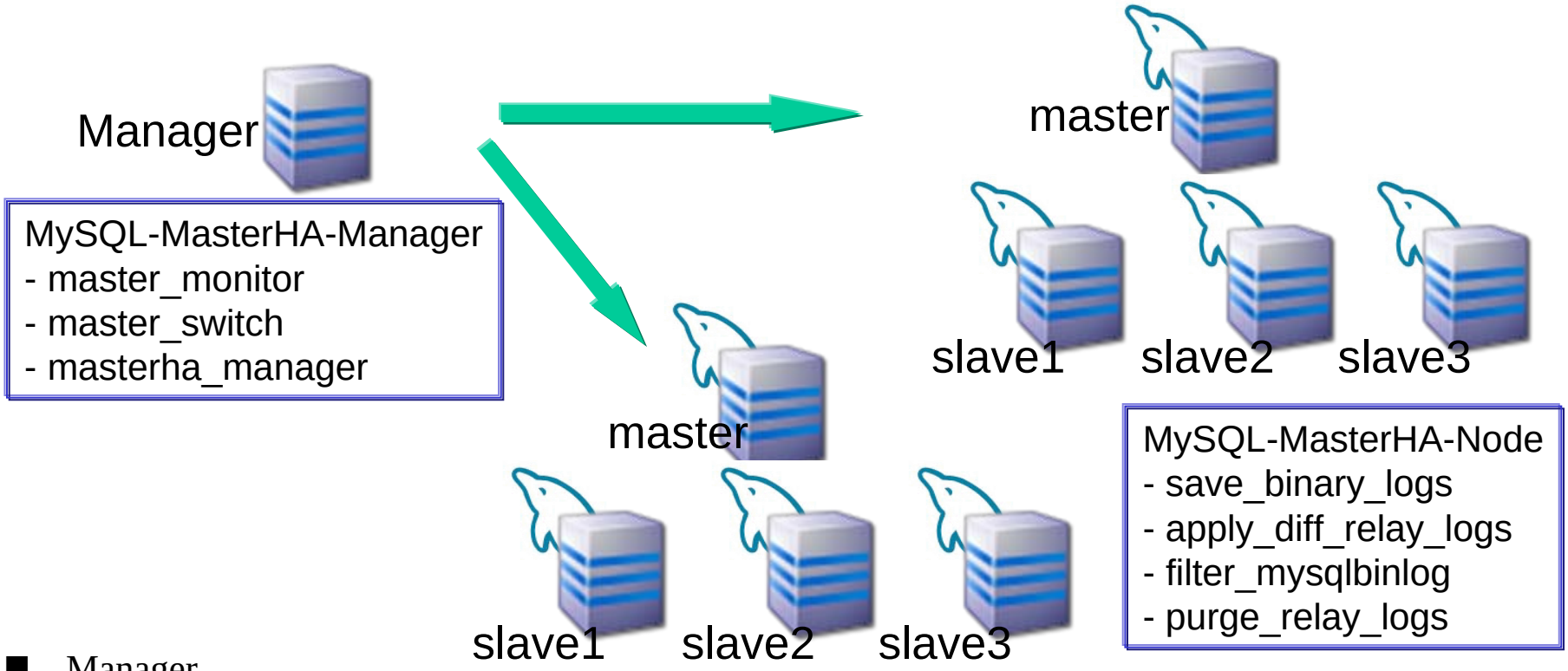
```
# at 2642668
# at 2642713
#110411 16:14:00 server id 1306  end_log_pos 2642713    Table_map: `db1`.`t1`
mapped to number 16
#110411 16:14:00 server id 1306  end_log_pos 2642764    Write_rows: table id 16
flags: STMT_END_F

BINLOG '
OKqiTRMaBQAALQAAABlTKAAAABAAAAAAAAAEABWdhbWVfAAJ0MQADAwP8
AQIG
OKqiTRcaBQAAMwAAAExTKAAAABAAAAAAAAAEAA//4CmgAAApoAAALAGFhY
WFhYTI2NjM0
'/*!*/;
```

- Multiple “#at” entries + same number of “end_log_pos” entries (when parsed by mysqlbinlog)
- “Table_map” event + “Write_rows (or others)” event + STMT_END
  - Write_rows events can be many when using LOAD DATA, Bulk INSERT, etc
- mysqlbinlog prints out when valid “Table Map .. STMT End” events are written
- If slave A has only partial events, it is needed to send complete “Table Map .. STMT End” events from the latest slave

Automating failover

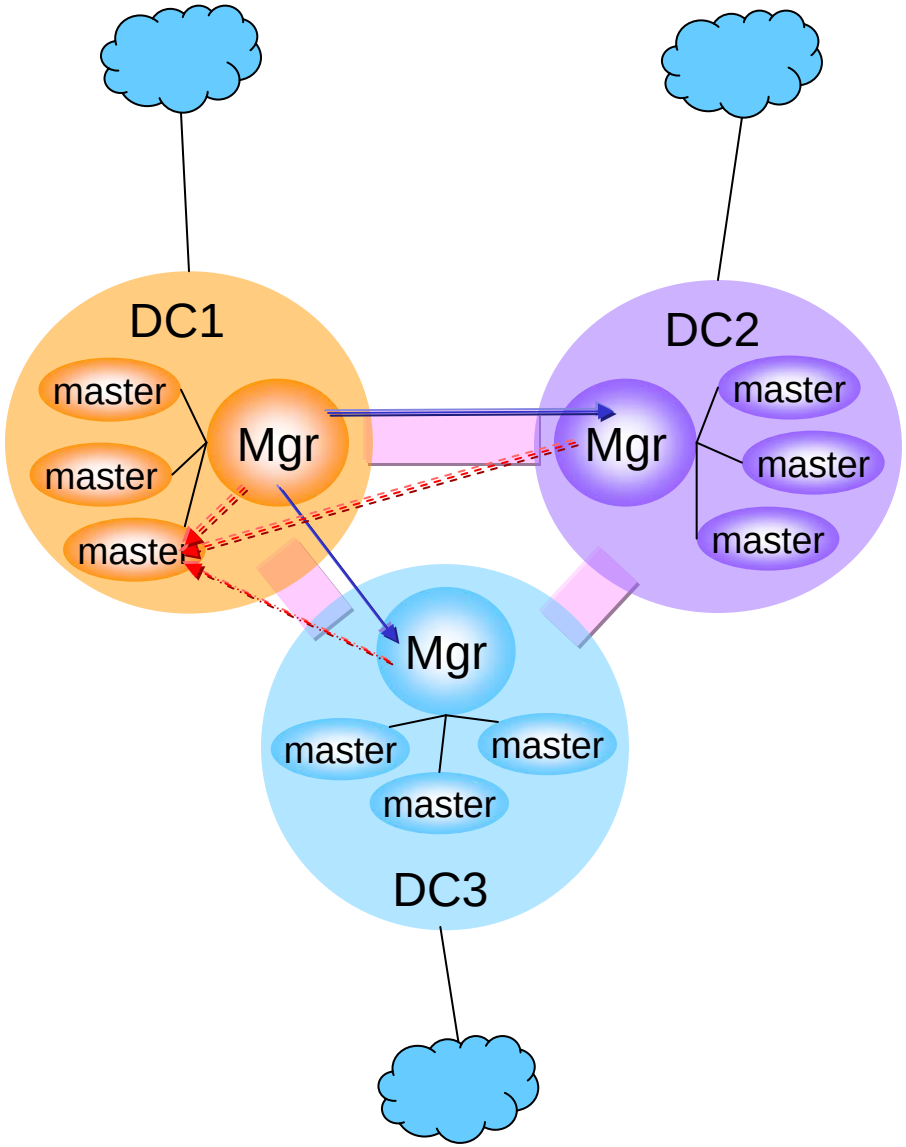
- Common HA tasks
  - Detecting master failure
  - Node Fencing (Power off the dead master, to avoid split brain)
  - Updating writer IP address
- Writing a script to do failover, based on what I have covered so far
- Running master failover scripts automatically
  - Make sure not to stop by stupid errors
    - Creating working/logging directory if not exists
    - Check SSH public key authentication and MySQL privileges at the beginning of starting the monitoring script
  - Decide failover criteria
    - Not starting failover if one or more slave servers are not alive (or SQL thread can't be started)
    - Not starting failover if the last failover has happened recently (within 8 hours)
- Notification/Operation
  - Sending mails
  - Disabling scheduled backup jobs on the new master
  - Updating internal administration tool status, master/slave ip address mappings, etc

Tool: Master High Availability Toolkit



- Manager
  - master_monitor: Detecting master failure
  - master_switch: Doing failover (manual, or automatic failover invoked by masterha_manager)
- Node : Deploying on all MySQL servers
  - save_binary_logs: Copying master's binary logs if accessible
  - apply_diff_relay_logs: Generating differential relay logs from the latest slave, and applying all differential binlog events
  - filter_mysqlbinlog: Trimming unnecessary ROLLBACK events
  - purge_relay_logs: Deleting relay logs without stopping SQL thread
- We have started using this tool internally. Will publish as OSS soon

One Manager per Datacenter



- Each Manager monitors multiple MySQL masters within the same datacenter
- If managers at DC2 and DC3 are reachable from the manager at DC1, and if a master is not reachable from none of the managers, the master failover procedure starts
  - Main purpose is to avoid split brain
- If any catastrophic failure (datacenter crash) happens, we do manual failover

Case

- Kernel panic happened on the master
- Checking whether the master is really dead (10 sec)
  - Checking SSH reachability (to check saving binlog is possible or not)
  - Check connectivity through other datacenters (secondary networks)
- STONITH (Forcing power off)
  - To make sure the master is really not active
  - Power off time highly depends on H/W
    - Dell PowerEdge R610: 5-10 seconds (via telnet+DRAC)
    - HP DL360: 4-5 seconds (via ipmitool+iLO)
- Master recovery
  - Finished in less than 1 second
- Parallel slave recovery
  - Finished in less than 1 second

Current limitations & tips

- Three or more-tier replication is not supported (i.e. Master->Master2->Slave)
  - Check Global Transaction ID project
    - Tracing differential relay log events becomes much easier
    - Binlog format needs to be changed (It doesn't work with -5.5)
- LOAD DATA [LOCAL] INFILE with SBR is not supported
  - It's deprecated actually, and it causes significant replication delay.
  - SET sql_log_bin=0; LOAD DATA … ; SET sql_log_bin=1; is recommended approach
- Replication filtering rules (binlog-do-db, replicate-ignore-db, etc) must be same on all MySQL servers
- Do not use MySQL 5.0.45 or lower version
  - end_log_pos is incorrect (not absolute): http://bugs.mysql.com/bug.php?id=22540
    - I did a bit hack to make the tool work with 5.0.45 since we still have some legacy servers, but generally upgrades should be done
  - When replication network failure happens, a bogus byte stream might be sent to slaves, which will stop SQL threads: http://bugs.mysql.com/bug.php?id=26489

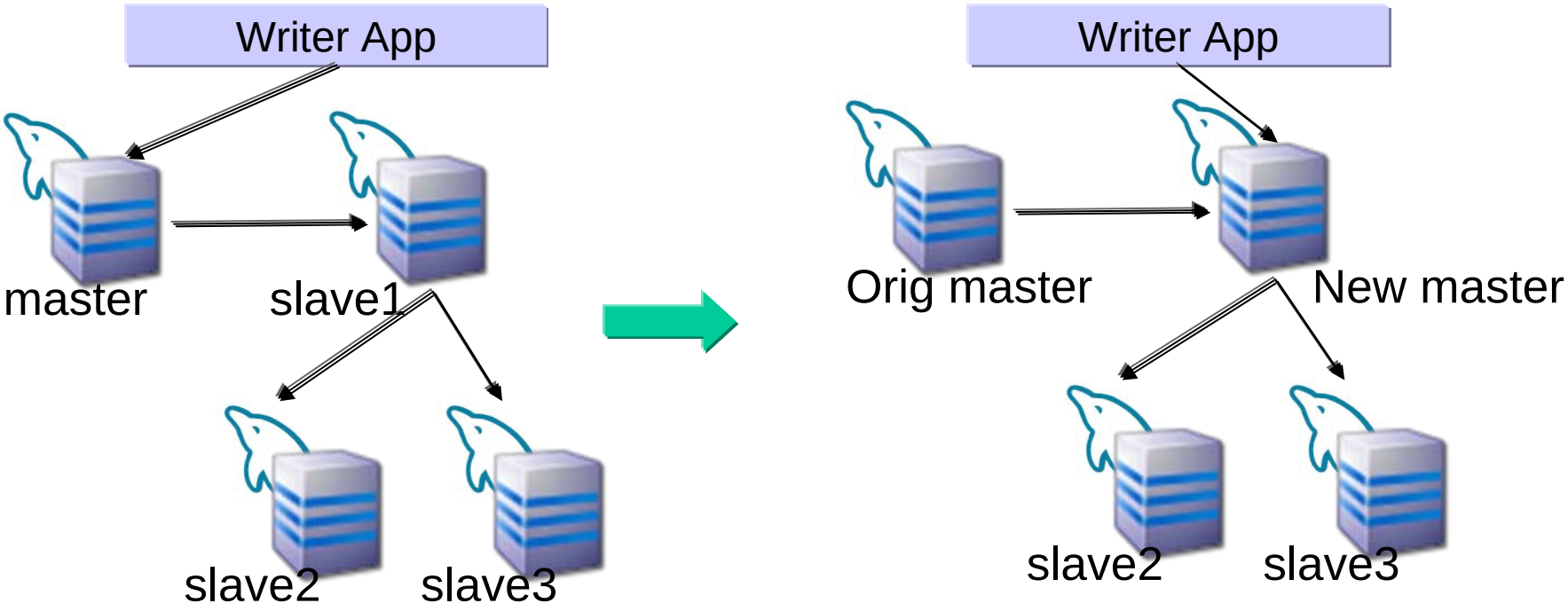
Table of contents

- Automating master failover
- Minimizing downtime at master maintenance

Minimizing downtime at master maintenance

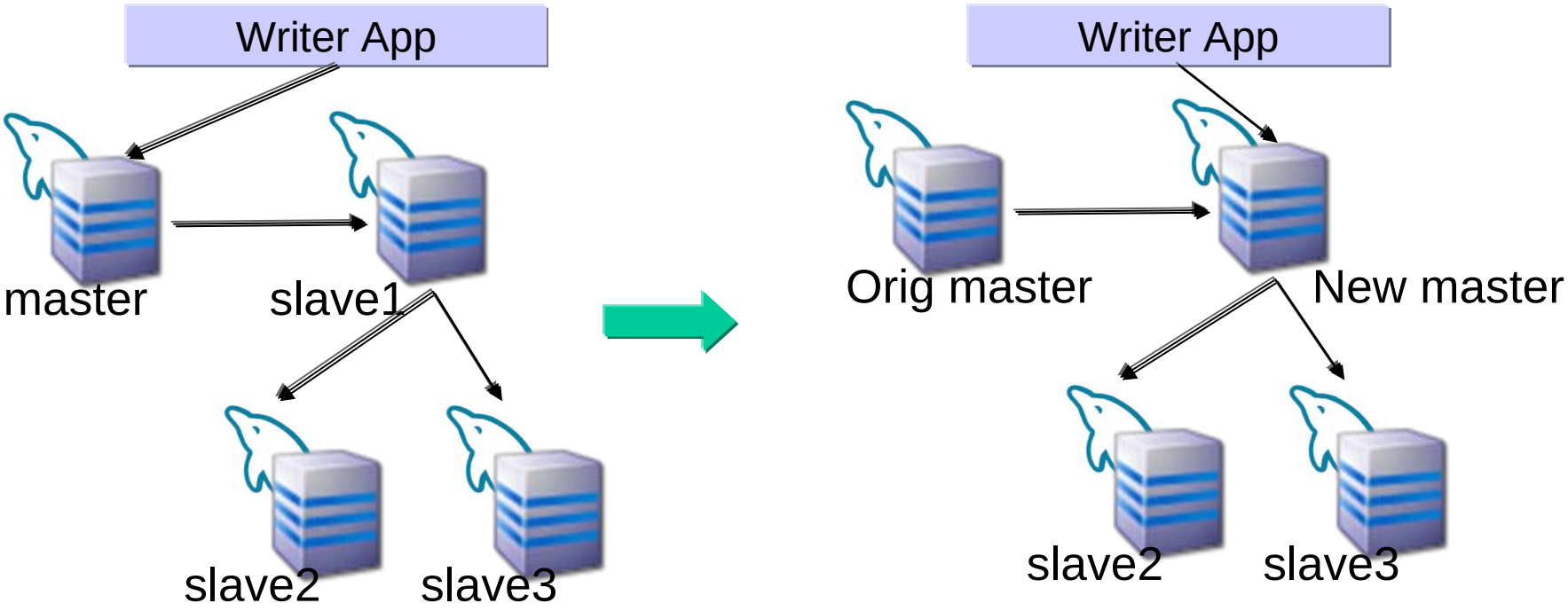
- Operations that need switching master
  - Upgrading MySQL
  - Replacing H/W components (increasing RAM, etc)
- Operations that do NOT need switching master
  - Adding/dropping index
  - Adding columns
  - oak-online-alter-table or Facebook OSC may help
- Adding/Changing shards
  - Can be done without stopping service, if designed well
  - Hash based sharding makes it difficult to re-shard without stopping services
  - Mapping table based sharding makes it much easier

Tentative three-tier replication



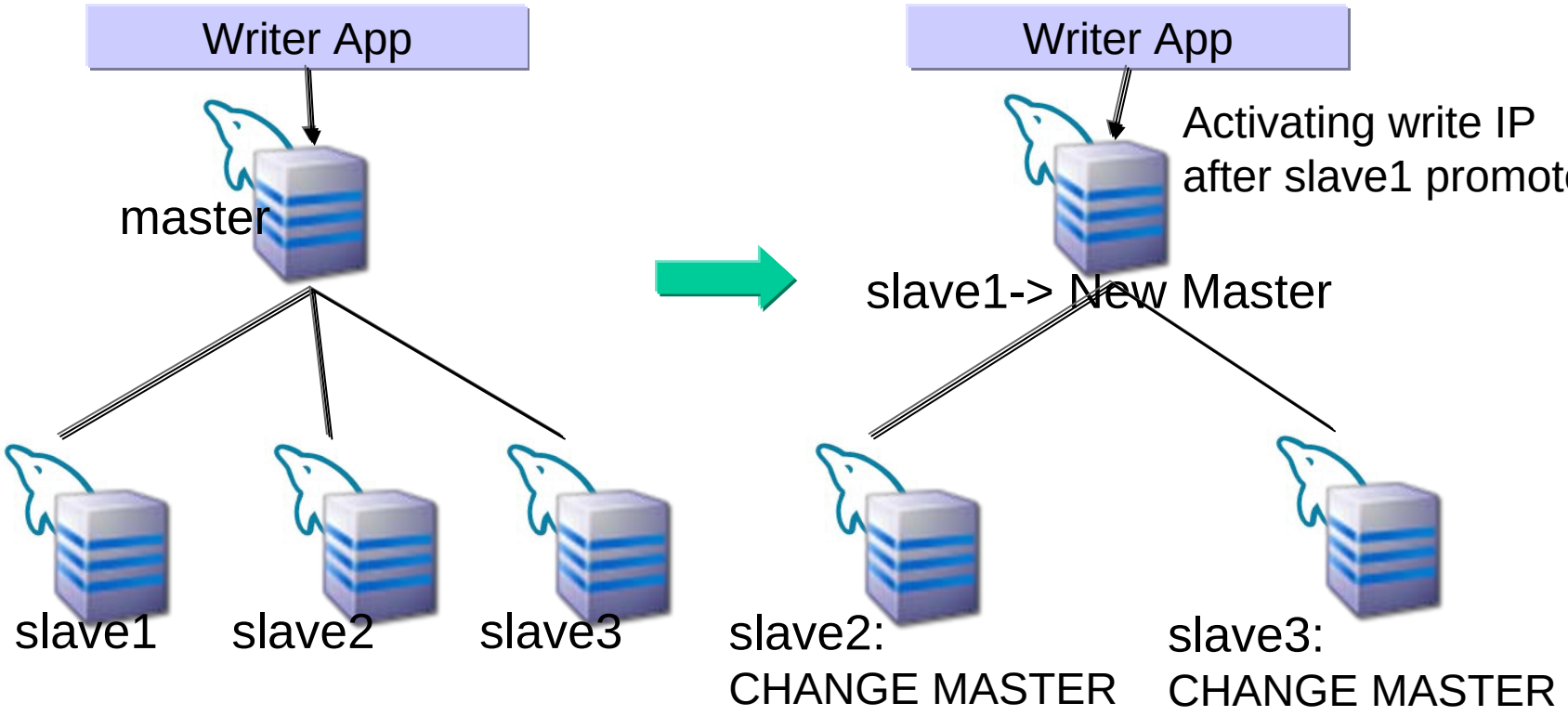
- Applications gradually establish database connections to the new master (or just moving writer IP address, if you can accept burst errors)
- Writes on the orig master will be finally replicated to the new master
- Destroying orig master when orig master has sent all binlog events

Tentative three-tier replication (2)



- Cons: Consistency problems might happen
  - AUTO_INCREMENT doesn't work (ID conflict), unless carefully using auto_increment_increment and auto_increment_offset
  - When the current master is updated, the row on the slave 1 is not locked
  - "#1. Updating cur master set value=500 where id=1, #2. Updating slave 1 set value=1000 where id=1, #3. Replicating #1" -> #2 is lost
  - Works well for INSERT-only, non-auto-inc query patterns
- Other possible approaches
  - Using Spider + VP storage engine on the orig master
    - Synchronous updates to the new master
    - Replicatoin channel must be disconnected between orig master and new master

Promoting one of slaves



- Cons: A few seconds of write downtime happens
  - Until slave 1 is activated as a new master
  - Master switch should be done as quickly as possible
    - Otherwise applications can not execute updates for a long time

Is a few seconds of downtime acceptable?

- In some cases it is acceptable for a few seconds of downtime on master
  - 500+ connections per second regularly
  - 100 connections at 3am
  - 2 seconds downtime will make 200 connections get tentative errors
    - Pushing reload button will be fine

Graceful master switch

- FLUSH TABLES WITH READ LOCK is not a silver bullet
  - Does not return errors immediately
  - Applications are kept waiting in orig master forever, unless read_timeout is set
  - Response time and number of connections are highly increased
- Updating multiple mysql instances (multiple shards) is not uncommon
  - “COMMIT Successful on node 1 -> COMMIT failure on node 2” results in data inconsistency
  - At least transaction commit should not be aborted
- More graceful approach
  - Rejecting new database connections (DROP USER app_user)
  - Waiting for 1-2 seconds so that almost all database connections are disconnected
  - Rejecting all updates except SUPER by SET GLOBAL read_only=1;
  - Waiting for .N second
  - Rejecting all updates by FLUSH TABLES WITH READ LOCK

Part of Master High Availability Toolkit

- "Fast master switch" functionality is included, mostly based on master failover tool
  - master_switch --master_state=alive
  - Master switch in 2 seconds (2 seconds for graceful writer blocks)

- Differences from master failover are:
  - Not automatic (interactive)
  - All servers must be alive
  - Replication delay must be zero
  - Freezing updates on the current master is needed
  - No power off
  - No binlog/relay log recovery (Just using MASTER_POS_WAIT() is fine)

Conclusion

- Automating master failover is possible
  - Without introducing too much complexity on application side
  - With 5.0/5.1 InnoDB
  - Without losing performance significantly
  - Without spending too much money
  - Works perfectly with Semi Synchronous Replication
- Our tool will soon be released as an open source software