Agenda

- Context
- Architecture
- Internals
- HA
Context

● **PolarDB is a cloud native DB offering**
  ○ Based on MySQL-5.6
  ○ Uses shared storage
  ○ Primarily for read scaleout
  ○ Also provides HA (multi DC HA using standby)

● **PolarDB uses:**
  ○ InnoDB as storage engine
  ○ InnoDB redo logs for physical replication
  ○ Supports shared storage Replica nodes and separate storage Standby nodes
Context

Terminology:

● Primary (aka Master): RW
● Slave
  ○ Replica: RO with shared storage
  ○ Standby: RO with separate storage (possibly in different DC)
    ■ Standby can have its own replicas

Goals:

● Ability to scale out dynamically
● HA (Zero data loss in case of master crash)
● Performance
Runtime Redo Application

● Moves Replica from one state to next
  ○ Apply Redo Logs generated on the primary (like recovery on the fly)
  ○ Redo Logs store physical page level changes

● Replication Lag == primary.written_lsn - replica.applied_lsn

● Minimize Replication Lag
  ○ For better service
  ○ For better performance
    ■ Flushing on primary (Design constraints)
    ■ Memory usage & redo application time on replica
Optimize Runtime Redo Application

- Better concurrency
  - Read Redo Logs (separate Async Reader thread)
  - Parse Redo Logs (single threaded)
    - Parse records
    - Store in multiple hash tables <space_id:page_no>
  - Apply Redo Logs
    - Multiple configurable LogWorker threads (innodb_slave_log_apply_worker)

- Multiple hash tables per worker thread
  - Avoid mutex contention
  - Efficient memory management
Optimize Runtime Redo Application

● InnoDB redo application code is written with one time, single threaded, startup recovery in mind

● Avoid double parsing
  ○ Store length of redo record
  ○ No need to parse the record when storing it to hash table

● Avoid rescanning
  ○ Start application from where we finished last time

● Use dummy indexes
  ○ Reusable index memory structures for redo apply
Optimize Runtime Redo Application

- Worker threads only work on cached pages
  - No extra IO for redo application
  - Freshly read in pages are updated in IO completion routine

- Do not apply batches atomically
  - Handle physical inconsistency on replica
  - No index level locking on replica to deal with page splits and merges
Dealing with Physical Inconsistency

- On primary multiple pages modified
  - Typically btree split or merge
- On replica multiple pages read
  - Typically range scan
- Add new log entry: MLOG_INDEX_LOCK_ACQUIRE
  - On replica register this by incrementing index::sync_counter
  - At mtr level:
    - If page is stale then close and reopen cursor
Dealing with Physical Inconsistency

- **Advantages:**
  - No system level locking for atomic batch application
  - No index level locking for page splits/merger
  - Only affected mtrs have to retry
  - No trx level retry
Flushed Constraints on Primary

- Replica cannot see a ‘too new’ page
  - For any freshly read block block.applied_lsn <= replica.applied_lsn
  - Implies that primary cannot write a block if block.newest_modification > replica.applied_lsn

- Hot page issue
  - block.newest_modification gets frequently updated
  - Primary unable to flush the page from flush_list
  - Primary can’t move forward buf_pool_oldest_modification
  - Checkpoint age keeps increasing
Flushed Constraints on Primary

- Pin well known hot pages in replica at startup
  - Primary is free to flush them
  - Doesn’t solve random hot page issue

- Copy hot pages on the primary
  - Once the copied page is flushable
    - Write it to disk
    - Move the block accordingly in the flush list
Primary

Flush List

<table>
<thead>
<tr>
<th>Oldest LSN</th>
<th>Newest LSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>160</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>160</td>
</tr>
</tbody>
</table>

Primary

Buffer Pool

<table>
<thead>
<tr>
<th>primary.write_lsn</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary.checkpoint_lsn</td>
<td>90</td>
</tr>
<tr>
<td>replica.applied_lsn</td>
<td>90</td>
</tr>
</tbody>
</table>
Torn Reads

- Read IO on replica when primary is writing same page
  - `innodb_replica_retry_page_read_times`
  - `innodb_replica_retry_read_wait`
MVCC

- InnoDB uses read_view and UNDO logs for MVCC
- read_view is an array of read/write trxs open when a trx starts
- Replica has no read/write trxs
  - No local read_view. Needs to know the open trxs on master at current applied_lsn
  - Initial read_view is sent by master as part of handshake
  - MLOG_TRX_START and MLOG_TRX_COMMIT entries to redo logs
- read_view on replica
  - Updated at redo apply batch boundary
  - Same read_view is shared amongst all trxs until applied_lsn is moved
Logical Consistency

- Non atomic redo application implies: `block::applied_lsn > replica::applied_lsn`
- How to avoid looking at ‘too new’ row version?
  - `read_view @ replica::applied_lsn` decides visibility
- How do we build the old version of the row?
  - By following `ROLL_PTR` in the row which points to UNDO page
- What if UNDO page has not yet been gone through redo application?
  - We’ll detect it and do it on the fly
- What if redo related to UNDO is not part of this batch
  - Not possible. InnoDB always log UNDO before actual data page
Purge

- Purge is garbage collection of free space
  - Clears up both data pages and UNDO pages
  - Reclaims deleted row space not visible to any other trx
- Purge read_view on primary built from
  - Oldest view on primary
  - Oldest view on replica
- Purge control
  - `innodb_primary_purge_max_lsn_lag`
  - `innodb_primary_purge_max_id_lag`
DDL

- Can’t touch tablespace on replica if the structure is being changed
  - DDL operations are synchronous
  - Table cache is invalidated

- **MLOG_META_CHANGE** to signify server level file operations
**HA: Adding a new Replica**

<table>
<thead>
<tr>
<th>Replica</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connects to master</td>
<td>Makes a checkpoint</td>
</tr>
<tr>
<td></td>
<td>Registers replica</td>
</tr>
<tr>
<td>Sends: oldest_lsn, newest_lsn,</td>
<td>Sends: oldest_lsn, newest_lsn, read_view, log file info (lsn, offset,</td>
</tr>
<tr>
<td>read_view, log file info</td>
<td>size)</td>
</tr>
<tr>
<td>Starts reading log from oldest_lsn</td>
<td></td>
</tr>
<tr>
<td>Parse and apply up to newest_lsn</td>
<td></td>
</tr>
<tr>
<td>Builds read_view</td>
<td></td>
</tr>
<tr>
<td>Goes online</td>
<td></td>
</tr>
</tbody>
</table>
HA: Failover to Replica

- Zero data loss
- No restart of replica (warmed up buffer pool)

Failover steps on replica:
  - Reopen files in rw mode
  - Change state to Standby
  - Apply redo to all pages (not just in the cache)
  - Flush pages to disk (now we have a flush_list)
  - Make full checkpoint
  - Change state to Primary
HA: Failover to Standby

- Failover steps on Standby:
  - Apply all redo logs up to the latest LSN
  - Reinitialize some in-memory structures like RSEG, change buffer etc.
  - Change state to Primary
  - Accept read/write workload
  - Rollback uncommitted txs
HA: RECOVER crashed Primary

- If we failover to Standby
  - New master can be behind crashed master
  - We want to avoid bootstrapping crashed master by copying all data

- RECOVER command
  - After crash recovery the old master
    - Sends a list of pages changed after failover LSN
    - Receives latest page images from new master
    - Directly write these pages to the disk
Questions?

Next Session: POLARDB for MyRocks - Make MyRocks Run on Shared Storage

Room E @ 3:00 PM