Visions and elements for MariaDB replication APIs

Kristian Nielsen
knielsen@askmonty.org

MariaDB Developer
Monty Program AB

MariaDB meeting, Istanbul, 2010

Outline

1. My long-term visions for replication
2. Elements of a replication API
3. How to implement the visions using the API

Outline

Visions

- Crash-safe replication
- Simple change of replication topology
  - Global transaction id . . .
- Group commit
- Synchronous replication (Galera)
- Parallel replication
- Cheaper durability
  - Less fsync(), less logs
- Pluggable replication (Tungsten, . . .)

Outline

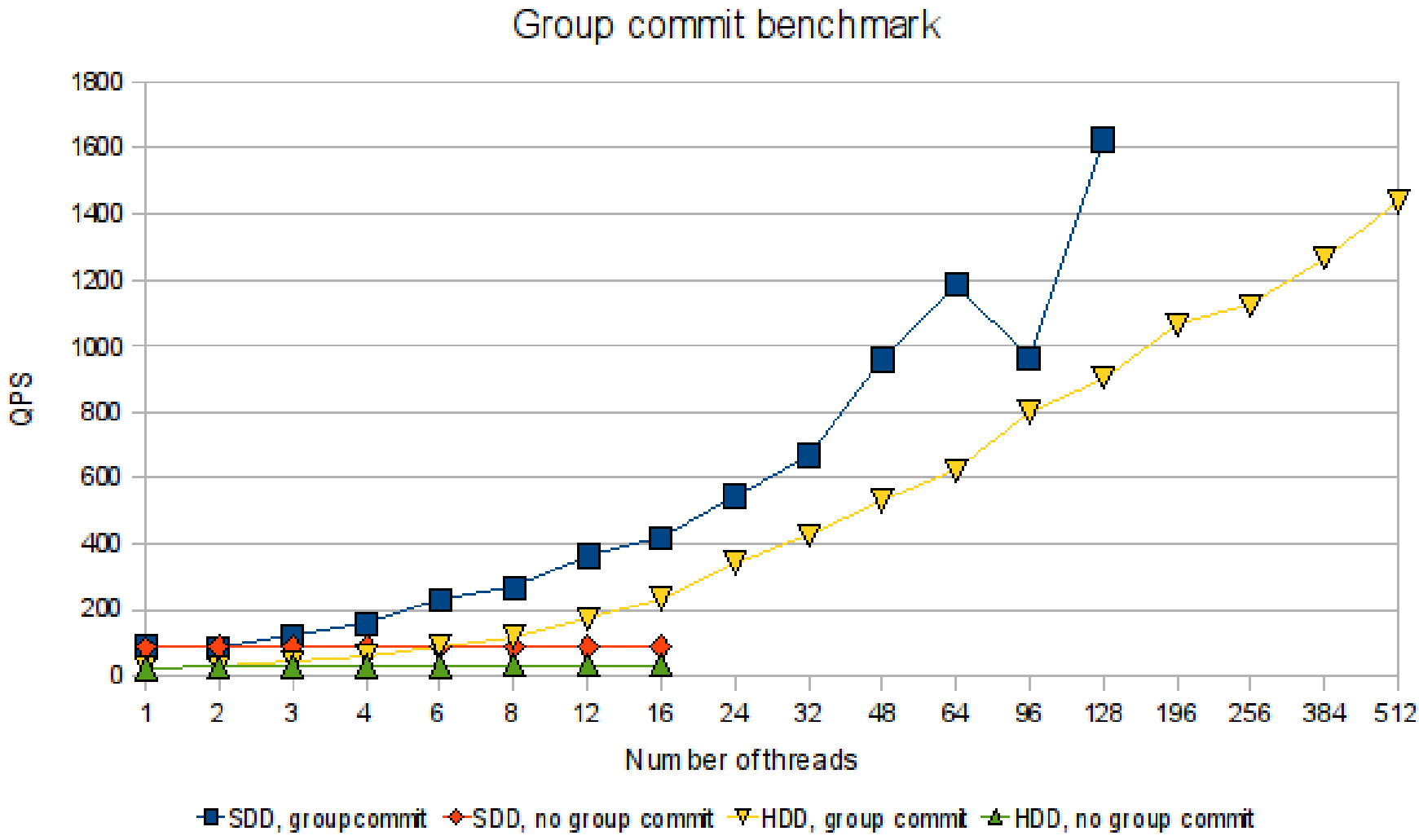
Transaction coordinator (TC) plugin [MWL#132]

- Does 2-phase commit between engines and binlog
- Recover into consistent state after crash

New API to allow plugin to take the role of TC

- Ability to use alternative binlog implementation
- Support group commit
- Provide consistent commit order between binlogs and engines
- Allow plugin to control commit order

Group commit benchmark



MariaDB

Group commit

Extends the storage engine API:

- Engine independent
- TC independent (allow alternative binlog plugin)
- Preserve consistent commit order
  - Obtain corresponding binlog position (without FTWRL)
  - Hot backups (XtraBackup, `mysqldump --single-transaction`)
  - Actually consistent START TRANSACTION WITH CONSISTENT SNAPSHOT
  - Safely release InnoDB locks early
  - Safe `--innodb-flush-log-at-trx-commit=0`, recovering lost transactions from binlog

Replication event generators [MWL#107]

- Hooks around the server to get all events that modify the database (INSERT, CREATE TABLE, etc.)
- Allow arbitrary plugin to subscribe, not just binlog
- Non-materialised API
  - Do not enforce a specific in-memory or in-disk format
  - Allow consumer to choose which information to use (eg. column index vs. name)
  - No least-common-denominator, no wasted copy of unneeded data
- Stacked generators to generalise binlog format
  - Row-based is stacked on top of statement
  - Want to support PBXT engine-level replication
  - Important to have lazy materialisation

Replication event applier [MWL#133]

- Generalisation of slave SQL thread
- Again non-materialised, "provider API"
  - Plugin supplies whatever information it has
  - Eg. accept either column name or column index
  - Error if insufficient information
- Clean way to create DDL and DML-capable background threads

Default materialised event format

- Eg. Google protobuf generation filter and event applier
- Make it easier to do simple plugins
  - Not require everyone to implement their own event format
- Enable eg. replication transports that are agnostic to underlying format
- Maybe can use existing binlog event format

Priority transactions

- Engines delegate the decision about how to handle parallel transactions that conflict
  1. Let second transaction wait (normal)
  2. Forcibly rollback first transaction (high-priority transactions)
- Not directly related to replication, but seems to be related
  - Galera needs it
  - I think also parallel replication could use it

Outline

How does this help Galera?

- Control commit order
  - Galera can implement a TC plugin
  - Control commit, including re-order or rollback transactions
    - Engine-independent
  - Guarantee consistent commit order (even cross-engine)
- Obtain primary key values
  - Galera needs it to detect conflicting transactions
  - Event generator API will provide this engine-independent

How does this help Galera?

- Obtain and apply events
  - Galera will want to use a default materialised event format
  - But also needs special information, like need for total order for DDL etc.
  - Stacked event model seems well suited here
- Priority transactions
  - Needed by the Galera replication algorithm

How does this help global transaction ID?

- Provide engine-independent and binlog-plugin-independent consistent commit order
- Engines can keep track of last local transaction ID
- Binlog implementation can map local transaction ID to global transaction ID
- Easily optain global transaction ID ("binlog position") from any server state.

How does this help crash-safety?

- Much can be done within current binlog (ie. global transaction ID)
- I think eventually a new binlog format is needed
  - Current one is not very extensible
  - Flush is expensive, and no protection against partial writes
  - Bad to put details like master file names and log rotations into events
- Keep current binlog as default catch-all
- Alternative implementations with more retrictions can be more performant and robust
  - Eg. disallow mixed InnoDB/MyISAM transactions.

Parallel replication

- New binlog implementation that writes transactions interleaved, but keeping COMMIT order
- Original parallelism from master kept intact
- Slave can safely execute interleaved events in parallel, as long as original COMMIT order is respected.
- Can speculatively execute T2 across T1 commit, but may need to rollback T2 in case of conflict
- Probably restricted to transactional MVCC engines

Conclusion

Informal discussion session after lunch

Questions?

Group commit benchmark

Details:

- Simple `REPLACE` query
- `innodb_flush_log_at_trx_commit=1`
- `sync_binlog=1`
- Binlog enabled and disabled
- Digital Western 10k HDD and Intel X25-M SSD

Observation:

- With binlog disabled, scales well with more threads due to group commit
- With binlog enabled, no scaling due to broken group commit