ORACLE

InnoDB Batch Commit Support for Optimized Group Commit and Replication Apply

Storage-engine primitive for coordinated commit

Nuno Carvalho

Replication Lead, Oracle

May 26, 2026

Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes 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 upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Goals for this discussion

The goal is to validate the feature contract before implementation details harden.

Get feedback on the feature and its source/replica use cases.

- Confirm the right semantics for atomic visibility and ordered persistence.
- Identify handlerton, InnoDB, binary log group commit, and replication apply integration risks.
- Get contribution and collaboration on semantics and implementation.

Problem statement

The server coordinates transaction groups, while the engine commits transactions one by one.

Source side

- Binary log group commit already forms a transaction batch.
- The storage-engine commit path repeats work for each transaction in the group.
- High-concurrency workloads leave commit-path efficiency on the table.

Replica side

- Very small transactions pay high metadata and commit overhead per transaction.
- Very large transactions create lag and reduce applier parallelism.
- Replica repositories and gtid_executed maintenance add write amplification beyond the source workload.

Requirements

Expose a minimal engine primitive that higher-level components can use safely.

Interface

- Accept a caller-supplied ordered batch of transactions.
- Make batch support discoverable before callers depend on it.
- Keep single-transaction commit as a trivial batch special case.

Semantics

- Atomic visibility: observers see either none or all effects from the batch.
- Ordered persistence: if a later transaction is durable, earlier batch transactions are durable too.
- Return one result for the batch: success or forced abort/rollback.

Scope control

- Reuse existing InnoDB commit logic as much as possible.
- Assume compatible transaction options or roll back/log errors in release builds.
- Atomic persistence remains a higher-level 2PC/recovery responsibility.

Challenges

The feature is simple to state, but subtle at visibility, persistence, and integration boundaries.

Visibility boundary

- No running transaction may observe a partial batch commit.
- Transactions inside the batch should not see each other before commit.
- Commit phases must be split without duplicating large parts of InnoDB commit code.

Persistence boundary

- Batch commit itself is not full all-or-nothing durability.
- Crash recovery may see a durable prefix and must rely on higher-level recovery integration.
- The caller-supplied ordering must be preserved on disk.

Caller expectations

- Source group commit and replica apply need a compatible API from day one.
- Replica reshaping may group small units or split large units, but final visibility must stay correct.
- The primitive must not force a source-only or replica-only redesign later.

Benefits: source commit path

Batch commit makes existing group commit information useful to InnoDB.

Use the group already formed

- Commit a full binary log group commit set as one coordinated engine batch.
- Align storage-engine commit work with the binary log pipeline.

Reduce repeated overhead

- Improve cache and scheduling behavior in the commit path.
- Make high-concurrency source commits more efficient.

Preserve semantics

- Each transaction remains logically independent.
- User-visible transaction results do not change.

Benefits: replication apply

Batch commit gives replication a safer foundation to reshape source workloads.

Small transactions

- Group small transactions into larger apply units.
- Reduce per-transaction metadata and commit overhead.

Large transactions

- Split very large transactions into independently prepared units.
- Prepare pieces in parallel when dependencies allow.

Replica throughput

- Improve parallel preparation and scheduling flexibility.
- Shorten lag created by inefficient transaction shape.

Benefits: design and operations

A shared primitive keeps source and replica improvements compatible from the start.

Independent delivery

- Source-side group commit optimization can deliver value first.
- Replica-side chunk batch commit can build on the same primitive later.

Operational recovery

- Shorten catch-up after backup restore, maintenance, or topology changes.
- Reduce resource consumption on replicas and leave more capacity for user workloads.

Long-term API

- Avoid a narrow source-only or replica-only handlerton API.
- Keep single-transaction commit behavior consistent as a batch special case.


ORACLE