Deep Dive: InnoDB Transactions and Write Paths

Marko Mäkelä

Lead Developer InnoDB

MariaDB plc

Comparing MariaDB Server to the ISO OSI Model

Open Systems Interconnection Model

7. *Application layer*
   - Example: HTML5 web application
   - Example: `apt update;apt upgrade`
6. *Presentation layer*
   - XML, HTML, CSS, …
   - JSON, BSON, …
   - ASN.1 BER, …
5. *Session layer*
   - SSL, TLS
   - Web browser cookies, …

Layers of MariaDB Server

7. *Client connection*
   - Encrypted or cleartext
   - Direct or via proxy
6. *SQL*
   - Parser
   - Access control
   - Query optimization & execution
5. *Storage Engine Interface*
   - `BEGIN`, `COMMIT`, `ROLLBACK`
   - Table cursors: open, read, write

MariaDB

Comparing MariaDB Server to the ISO OSI Model

Open Systems Interconnection Model

4. *Transport layer*
   - TCP/IP streams out of IP packets
   - Retransmission, flow control
3. *Network layer*
   - router/switch
   - IP, ICMP, UDP, BGP, DNS, …
2. *Data link layer*
   - Packet framing, checksums
1. *Physical layer*
   - MAC: CSMA/CD, CSMA/CA, …
   - Ethernet, ATM, RS-232, WiFi, …

Layers of MariaDB Server

4. *InnoDB Transaction* (undo log)
   - Atomic, Consistent, Isolated access to tables via Locks & Read Views
   - Distributed transactions (XA 2PC)
3. *InnoDB Mini-transaction* (redo log)
   - Atomic, Durable multi-page changes
   - Page checksums, crash recovery
2. *Operating & File System* (block cache)
   - ext4, XFS, NFS, NTFS, ReFS, …
1. *Hardware/Firmware* (physical storage)
   - HDD, SSD, NVMe, CXL.mem, …

MariaDB

Diving Deep into Blocks

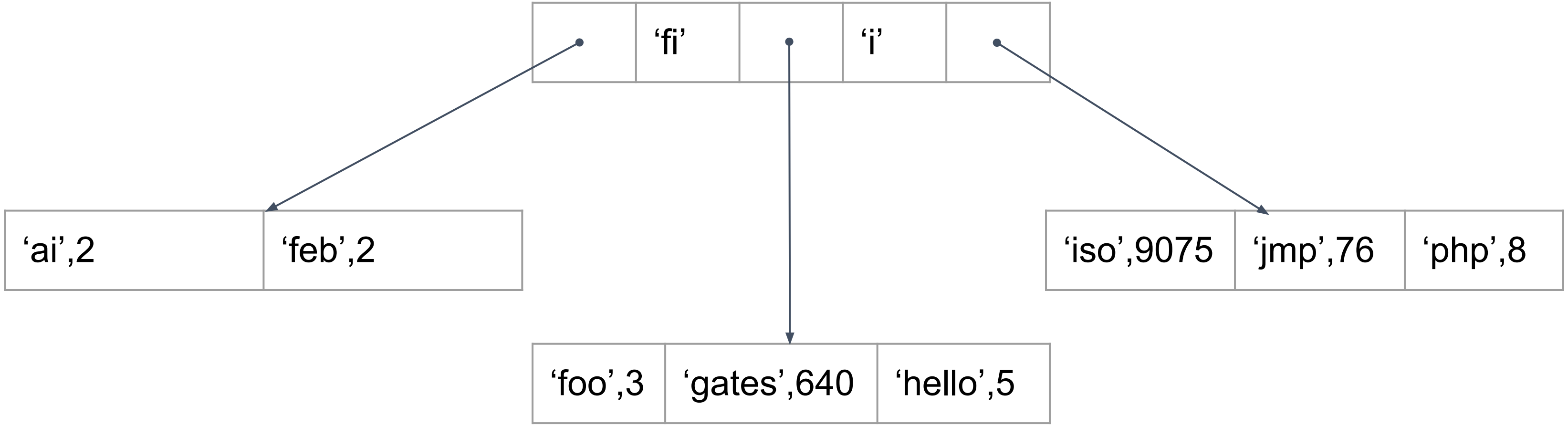
MariaDB

Storing Tables in Files of Fixed-Size Blocks (Pages)

- Tables can be viewed as a collection of data records and indexes.
  - Heap organized (MyISAM, Aria): Records are stored separately in a heap, and indexes contain keys along with heap positions (rowid).
  - Index organized (InnoDB): Clustered index records comprise the `PRIMARY KEY` and ~~system~~ and user columns. Secondary records include the key and `PRIMARY KEY`.
- `CREATE TABLE t(`**`a`**` INT PRIMARY KEY, `**`b`**` INT UNIQUE, `*`c`*` INT);`
  - MyISAM: heap (rowid,*a*,*b*,*c*) and indexes (**a**,rowid), (**b**,rowid)
  - InnoDB: indexes (**a**,~~`DB_TRX_ID`~~,~~`DB_ROLL_PTR`~~,*b*,*c*) and (**b**,**a**)

B-tree Basics

- B-tree is a popular page oriented implementation of indexes.
  - Starting from the root page there are *n* ordered keys (or key prefixes or parts) and *n*+1 pointers to lower-level (child) pages.
- At the bottom we have *leaf* pages that contain entire records.
- Example: `CREATE TABLE t(a VARCHAR PRIMARY KEY,b INT);`



MariaDB

Some Data Structures in InnoDB Tablespace Files

- Allocation bitmaps (at page *n*·`innodb_page_size`), file segments (lists of pages belonging to a “subfile”, such as `SEG_TOP`, `SEG_LEAF` of an index)
- Tables: Collections of index trees, each starting at an immovable root page
  - In `*.ibd` files, the clustered index root page always is 3.
- Core data dictionary tables starting at hard-coded pages in the system tablespace: `SYS_TABLES`, `SYS_COLUMNS`, `SYS_INDEXES`, `SYS_FIELDS`
- Transactions: The `TRX_SYS` page points to undo log header pages, which point to pages of undo log records of uncommitted or to-be-purged transactions. They are in undo tablespaces (`undo001`, `undo002`, …) or in the system tablespace.

MariaDB

Mini-Transaction Layer

MariaDB

InnoDB Mini-Transaction Layer

A **mini-transaction** manages an **atomic** set of page reads or writes, covered by **page latches** (or **buffer-fix**) and **write-ahead log** (redo log) for **durability**.

- An ”optimistic insert” of a record updates several parts of a B-tree leaf page.
- A ”pessimistic insert” will add at least 1 page to the B-tree, splitting the leaf.
- A mini-transaction can **modify at most one index** B-tree (and 1 undo page)!
- A mini-transaction can **read from 2 indexes**: secondary, clustered, (undo)
    - Allowing atomic modification of clustered, secondary would lead to *lock order inversion* and *deadlock* between (say) concurrent `INSERT` and `SELECT`.
    - Undo log pages are never overwritten; a buffer-fix will suffice for reads.

InnoDB Mini-Transaction Layer Example: `INSERT`

A single-row `INSERT` transaction involves several mini-transactions:

1. Write undo log and optimistically insert a record into clustered index.
    - If a page split is needed, run another mini-transaction for a pessimistic insert.
2. Insert a record into the first secondary index.
    - If a page split is needed, run another mini-transaction for a pessimistic insert.
3. (Insert a record into each subsequent next secondary index.)
4. Mark the transaction state as committed in the undo log header.

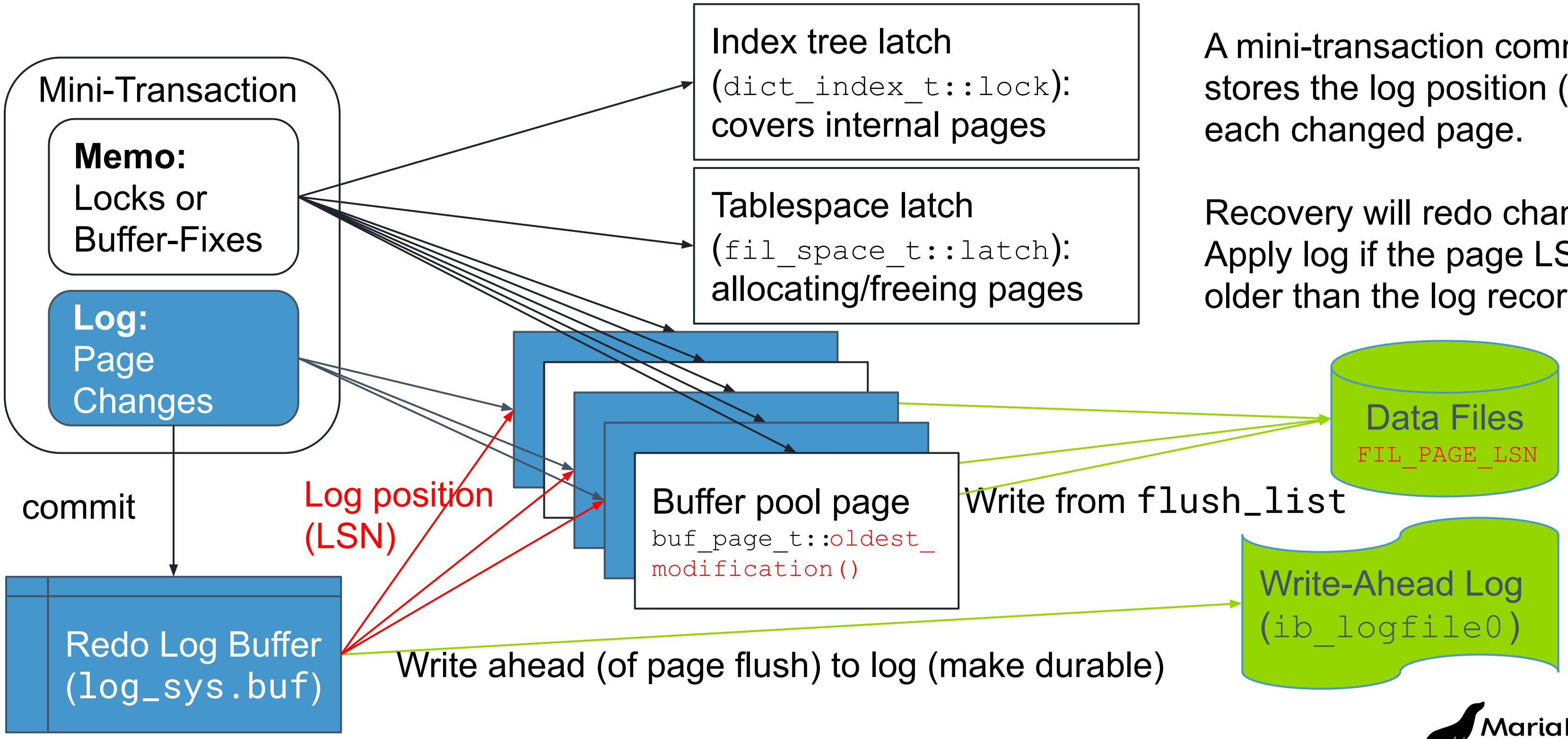
InnoDB Mini-Transaction Recovery

- ”The log is the database”: the data pages are basically just a cache of it.
- The write-ahead log (redo log) defines the state of all persistent InnoDB data pages, at a specific point of logical time (log sequence number, LSN).
- A **checkpoint truncates the start of the log** to save space and startup time, after ensuring no unwritten page changes are older than the checkpoint LSN.
- Recovery will apply the log from the latest checkpoint LSN up to the end.
- The `ib_logfile0` is circular; ”the end” is defined by discontinuity.
- The pages will be recovered to match the last complete mini-transaction.
    - Persistent data pages never are newer than the **write-ahead log**!

Understanding Mini-Transactions and Recovery

A mini-transaction commit stores the log position (LSN) to each changed page.

Recovery will redo changes: Apply log if the page LSN is older than the log record LSN.



MariaDB

InnoDB Transaction Layer

MariaDB

InnoDB Transaction Layer

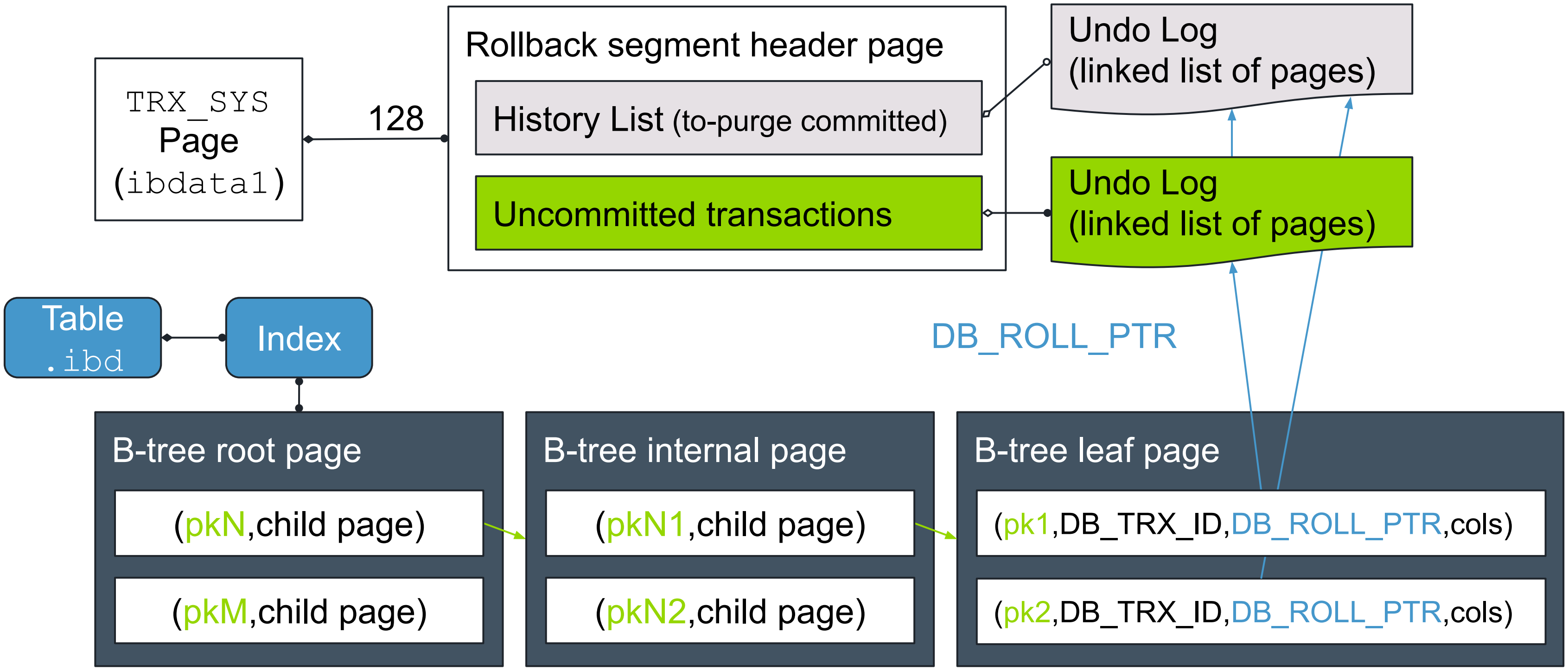
- *Optimistically* checks locks and changes data on the go; **transaction commit never fails**, but a **rollback is expensive** (revert the undo log and commit).
- Relies on atomic mini-transactions writing one undo log record or updating one index at a time, or changing the state of a transaction.
- On startup, any pending (incomplete) transactions will be recovered from undo logs, along with their implicit exclusive locks on any modified records.
- Automatic rollback in the background, except for `XA PREPARE`, which will wait for explicit `XA COMMIT` or `XA ROLLBACK` from the client (or binlog).
- InnoDB supports **non-locking read** (multi-versioning concurrency control) by read view '**snapshots**' that are based on undo logs.

Reaching ACID (Atomic, Consistent, Isolated, Durable)

- A transaction writes **undo log** before modifying indexes.
  - Usually row-level: insert/update a record with this primary key.
  - Bulk insert into an empty table may write just ”truncate on rollback”.
- A **read view** for **snapshot isolation** identifies the set of committed transactions at its creation. Non-**locking** reads may access undo logs of other transactions.
- Once no read view needs the undo logs of some committed transaction, they may be **purged**, along with records that have been updated to delete-marked. Purge also covers DDL recovery, such as `unlink()` after `DROP TABLE`.

MariaDB

A Page View of the InnoDB Transaction Layer



MariaDB

How Transactions Spray ACID on Mini-Transactions

- Transactional **locks** and **write-ahead *undo* log** are the glue for making the operations on the rows of tables appear atomic and consistent.
- Modifying a row in a multi-index table is not atomic! MVCC, purge and lock checks in a secondary index must look up the `DB_TRX_ID` via `PRIMARY KEY`. (To be improved in MDEV-17598.)
- MVCC and purge may retrieve undo log based on `DB_ROLL_PTR` and construct an older version until a visible `DB_TRX_ID` is found.
- `BLOB` (or `TEXT` or long `VARCHAR`) copy-on-write is not atomic; even an UPDATE may move *unaffected* columns off-page; see MySQL Bug #62037.
- `READ UNCOMMITTED` transactions may see truncated BLOB contents.

Finding the Right Balance

Finding the Right Level of Atomicity and Concurrency

- Page latches and row-level locks allow quite some concurrency. Not perfect:
- Before MariaDB 10.3, separate mini-transactions created a transaction and wrote its first undo record. Pointless and expensive, invisible to others.
- `UPDATE` and `DELETE` first execute a locking read, which will create an explicit lock and release the page latch. MDEV-16232 would allow `INSERT`-style implicit locking based on `DB_TRX_ID` in the record.
- MariaDB can release unmodified pages in a mini-transaction, as well as retain a buffer-fix to speed up cases that need mini-transaction restart (MDEV-34791).
- Deadlocks due to lock order inversion cannot occur in no-wait cases. Example: MDEV-37115: “trylock” previous page to optimize reverse index scan.

MariaDB

Performance Oriented Data Structure Changes

- 10.3: Read-only `TRX_SYS` page and lock-free `trx_sys.rw_trx_hash` table. Merged `insert_undo` and `update_undo` into a single log.
- 10.5: More compact, easier-to-parse log record format. Custom allocator for recovery. `FREE_PAGE` records avoid write-back of garbage pages. Doublewrite buffer is skipped for (re)initialized pages. All page writes are asynchronous, allowing concurrent fdatasync().
- 10.6: Rewritten latches and locks. Copy-free undo log access via buffer-fixed pages. Purge coordinator looks up all tables, concurrently with undo truncation.
- 10.8: Variable log block size allows concurrent writes to `log_sys.buf`; each writer thread encrypts its own log and computes CRC-32C before writing.

Logical and Physical Logging in InnoDB

Logical Undo Log

- Tables are identified by `SYS_TABLES.ID`; reassigning it ”detaches” old log
- Indexes on virtual columns are identified by `SYS_INDEXES.ID`
- Records are identified by the values of `PRIMARY KEY` and updated fields
- `BLOB` pointers are physical

Physical Redo Log (`ib_logfile0`)

- Type, length, and value starting with (tablespace_id,page_number).
- Recovery will find files based on `FILE_DELETE`, `FILE_RENAME`, `FILE_MODIFY` records.
- To reduce log volume, some operations (insert a record) use partially logical log format.

The Circular InnoDB Write-Ahead Log `ib_logfile0`

- The preallocated file allows fast in-place writes (with `O_DIRECT` in 10.11+).
- Recovery is possible if all records since the latest checkpoint fit in the file (**the log must not overwrite itself**, or the “checkpoint age” must be small enough).
- A too small `innodb_log_file_size` causes frequent writes of pages from the buffer pool to advance the checkpoint (write amplification).
- Up to MariaDB Server 10.6, log records are split into 512-byte blocks:
  - One writer at a time appended its records and updated the log block checksum, while hogging `log_sys.mutex` and blocking other writers.
  - `mariadb-backup --backup` could easily keep up, using a simple log block parser.

Performance oriented log format changes (FOSDEM 2022)

- Make each mini-transaction (10 B to 2 MiB) a logical block on its own
  - Log block header shrunk from 96 bits to a 1-bit sequence number.
  - The sequence bit flips each time the circular file ”wraps around”, allowing recovery to detect the end of log (old garbage written before the latest checkpoint).
- Truly concurrent execution of multiple `mtr_t::commit()`:
  - Concurrent threads compute their own CRC-32C before knowing the LSN.
  - Concurrent `memcpy()` to `log_sys.buf` is covered by *shared* `log_sys.latch`.

Optimizing LSN Allocation (MDEV-33515, MDEV-21923)

- Concurrent writes to `log_sys.buf` are protected by *shared* `log_sys.latch`.
  - How to allocate non-overlapping slices of `log_sys.buf` and the LSN?
  - Original solution: a `log_sys.lsn_lock` protecting several fields
- Better: A “critical section” of just `fetch_add(size + WRITE_TO_BUF)`
  - Accurately updates `Innodb_log_writes` (some users are obsessed with counters).
  - Reads and advances the base of the LSN and the buffer offset by the needed size.
  - Reads also a `WRITE_BACKOFF` flag (indicating that special handling is needed).

MariaDB

The Optimized LSN Allocation Logic (MDEV-21923)

```
while (UNIV_UNLIKELY((l= write_lsn_offset.fetch_add(size + WRITE_TO_BUF) &
                     (WRITE_TO_BUF - 1)) >= buf_size))
{
  /* The following is inlined here instead of being part of
  append_prepare_wait(), in order to increase the locality of reference
  and to set the WRITE_BACKOFF flag as soon as possible. */
  bool late(write_lsn_offset.fetch_or(WRITE_BACKOFF) & WRITE_BACKOFF);
  /* Subtract our LSN overshoot. */
  write_lsn_offset.fetch_sub(size); /* one call parameter less below */
  append_prepare_wait(late, ex); /* ensures WRITE_BACKOFF is cleared */
}
// set_for_checkpoint() logic omitted
return {l + base_lsn.load(std::memory_order_relaxed), l + buf};
```

x86-64 `LOCK XADD`

x86-64 `LOCK SUB`

x86-64 `LOCK BTS`

Main Uses of *Exclusive* `log_sys.latch`

- `log_sys.get_lsn()`; MDEV-21923 removed `log_sys.lsn`
  - Lock-free `log_sys.get_lsn_approx()` for adaptive flushing heuristics
- Any writes to `ib_logfile0` must shortly freeze the source (`log_sys.buf`).
  - `log_t::write_buf()`: reset `write_lsn_offset`, `swap(buf, flush_buf)`
  - Outside DDL or checkpoint, `log_sys.latch` is **released before file system access to allow concurrent writes** to the swapped `log_sys.buf` to resume.

File operations on `ib_logfile0` are covered by `write_lock`, `flush_lock` in the group commit implementation (MDEV-21534, MDEV-24341, MDEV-26789).

Some Notable Changes in the InnoDB Buffer Pool I/O

- MDEV-27774 removed `buf_pool.flush_order_mutex`.
    - We must sort when inserting first-time-dirtied blocks to `buf_pool.flush_list`.
        - Modifying a previously clean page should be rather rare.
    - `flush_rbt` was removed long ago in MDEV-23399; recovery would sort too.
- MDEV-19738: skip doublewrite for freshly (re)initialized pages
- MDEV-29911: parallel “fake read” threads recover pages based on log records
- MDEV-25948: “bleeding edge” doublewrite, disregarding write-ahead logging
- MDEV-35609 (future task): make use of Linux 6.13 atomic writes

MariaDB

THANK YOU