Deep Dive: InnoDB Transactions and Write Paths

From the client connection to physical storage

Marko Mäkelä, Lead Developer InnoDB
Michaël de Groot, MariaDB Consultant

Second Edition, for
MariaDB Developers Unconference, Tampere, Finland
June 28, 2018

InnoDB Concepts

Some terms that an Advanced DBA should be familiar with

A **mini-transaction** is an atomic set of page reads or writes, with write-ahead **redo log**.

A **transaction** writes **undo log** before modifying **indexes**.

The **read view** of a transaction may access the undo logs of newer transactions to retrieve old versions.

Purge may remove old undo logs and **delete-marked records** once no read view needs them.

The “storage stack” of InnoDB

A Comparison to the OSI Model (Upper Layers)

The 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, …

Some layers inside the MariaDB Server:

7. client connection

- Encrypted or cleartext
- Direct or via proxy

6. SQL

- Parser
- Access control
- Query optimization & execution

5. Storage engine interface

- Transactions: start, commit, rollback
- Tables: open, close, read, write

A Comparison to the OSI Model (Lower Layers)

The Open Systems Interconnection Model

4. Transport layer
- TCP/IP turns packets into reliable streams
- Retransmission, flow control

3. Network layer
- router/switch
- IP, ICMP, UDP, BGP, DNS, …

2. Data link
- Packet framing
- Checksums

1. Physical
- MAC: CSMA/CD, CSMA/CA, …
- LAN, WLAN, ATM, RS-232, …

InnoDB Storage Engine

4. Transaction
- **Atomic**, **Consistent**, **Isolated** access to multiple tables via Locks & Read Views
- XA 2PC (distributed transactions by user, or binlog-driven for cross-engine commit)

3. Mini-transaction
- **Atomic**, **Durable** multi-page changes
- Page checksums, crash recovery

2. Operating System (file system, block device)
- Ext4, XFS, ZFS, NFS, …

1. Hardware/Firmware (physical storage)
- Hard disk, SSD, NVRAM, …

SQL and storage engine interface

Step 1: SQL Layer

UPDATE talk SET attendees = 25 WHERE conference="M|18" AND name="Deep Dive";

- Constructs a parse tree.
- Checks for permissions.
- Acquires metadata lock on the table name (prevent DDL) and opens the table.
- Retrieves index cardinality statistics from the storage engine(s).
- Constructs a query execution plan.

Step 2a: Read via the Storage Engine Interface

UPDATE talk SET attendees = 25 WHERE conference="M|18"
AND name="Deep Dive";

- Find the matching record(s) via the chosen index (secondary index or primary key index), either via lookup or index scan.

- On the very first read, InnoDB will lazily start a transaction:
    - Assign a new `DB_TRX_ID` (incrementing number global in InnoDB)
    - No read view is needed, because this is a locking operation.

Step 2b: Filtering the Rows

UPDATE talk SET attendees = 25 WHERE conference="M|18"
AND name="Deep Dive";

- If the WHERE condition can only be satisfied by range scan or table scan, we will have to filter out non-matching rows.
  - If Index Condition Pushdown is used, InnoDB evaluates the predicate and filters rows.
  - Else, InnoDB will return every record one by one, and the SQL layer will decide what to do.
- After returning a single record, a new mini-transaction has to be started
  - Repositioning the cursor (search for a key) is moderately expensive
  - Optimization: After 4 separate reads, InnoDB will prefetch 8 rows (could be improved)

Step 2c: Locking the rows

- InnoDB will write-lock each index leaf record that was read
  - Note: InnoDB has no “table row locks”, but “record locks” in each index separately.
  - Condition pushdown would allow to avoid locking non-matching records.
- All records that may be changed are locked exclusively
  - When using a secondary index, all possibly matching records stay locked
  - Depending on the isolation level, InnoDB may unlock non-matching rows in primary key scan (MySQL Bug #3300, “semi-consistent read”)
  - Updates with a `WHERE` condition on `PRIMARY KEY` are faster!

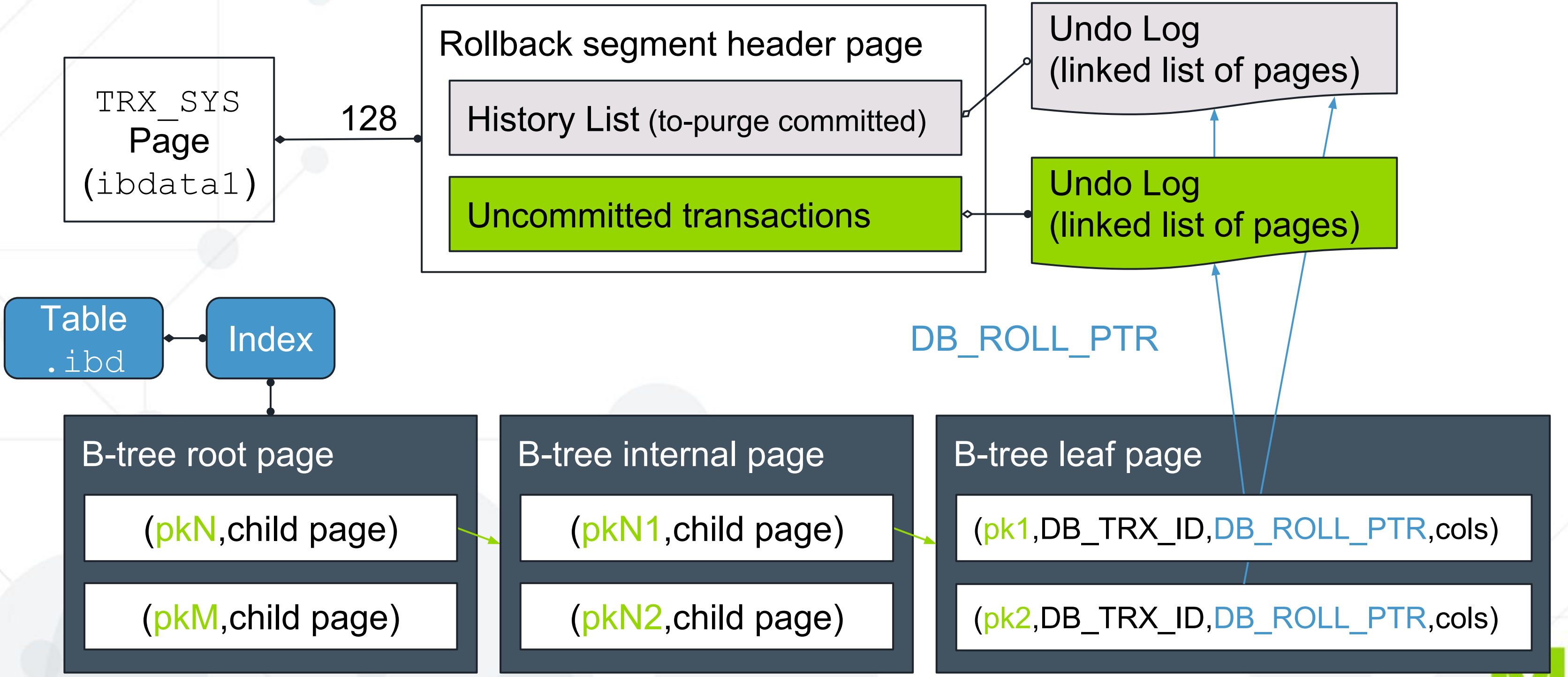
Transaction layer

InnoDB Transaction Layer

The InnoDB transaction layer relies on atomically updated data pages forming persistent data structures:

- Page allocation bitmap, index trees, undo log directories, undo logs
- Undo logs are the glue that make the indexes of a table look consistent.
- On startup, any pending transactions (with implicit locks) will be recovered from undo logs.
  - Locks will be kept until incomplete transactions are rolled back, or
  - When found in binlog after InnoDB recovery completes, or
  - until explicit `XA COMMIT` or `XA ROLLBACK`.
- InnoDB supports non-locking reads (multi-versioning concurrency control) by read view ‘snapshots’ that are based on undo logs.

A Page View of the InnoDB Transaction Layer



DB_TRX_ID and DB_ROLL_PTR

- DB_TRX_ID is an increasing sequence number, global in InnoDB
  - Each user transaction (except read-only non-locking) gets a new one, upon **transaction create**
  - While other transaction ids (GTID, binlog position) are in commit order, this transaction id is in create order
  - Each record has metadata with its current DB_TRX_ID
  - If the DB_TRX_ID is in a list of uncommitted transactions, there is an **implicit lock**
- DB_ROLL_PTR is a pointer to the undo log record of the previous version
  - Stored in the record's metadata (hidden column in the clustered index)
  - Or a flag in it is set, meaning that there is no previous version (the record was inserted)

Step 3: Updating the Matching Rows

```
UPDATE talk SET attendees = 25 WHERE conference="M|18"
  AND name="Deep Dive";
```

- update_row(old_row,new_row) is invoked for each matching row
- InnoDB calculates an "update vector" and applies to the current row.
- The primary key (clustered index) record will be write-locked.
  - It may already have been locked in the "Read" step.
- An undo log record will be written in its own mini-transaction.
- The primary key index will be updated in its own mini-transaction.
- For each secondary index on updated columns, use 2 mini-transactions:

Step 3a: Creating undo log records

- On the first write, the transaction is created in the buffer pool:
  - Create or reuse a page (same size as other pages)
  - Create undo log header, update undo log directory (also known as “rollback segment page”).
  - Done in separate mini-transaction from undo log record write; optimized away in MariaDB 10.3
- Undo log pages are normal data pages: They can fill up and get persisted
- All changes of the same transaction are added to the same undo log page(s)
- *Writing each undo log record* means (in a mini-transaction):
  - Append to last undo page, or allocate a new undo page and write the record there.
  - InnoDB writes each undo log in advance, before updating each index affected by the row operation.
- Undo log records need to be purged when they are no longer needed

Step 3b: Modifying the Indexes

- After the undo log record has been written, the modification is applied to the clustered index (PRIMARY KEY) and any affected secondary indexes
  - Updating PRIMARY KEY will cause the record to be delete-marked and a new record to be inserted, in every index.
- Clustered index: other than PRIMARY KEY columns are updated in place.
  - If the record size changes, pages may be split or merged.
  - BLOBs (copy-on-write) are written as linked list of pages, 1 page per mini-transaction.
- Each index is modified in a separate mini-transaction.
- In secondary indexes, UPDATE is 2 mini-transactions:
  - (1) search, lock, delete-mark old record, (2) insert new record.

Step 4a: Commit

UPDATE talk SET attendees = 25 WHERE conference="M|18"
AND name="Deep Dive";

- Because autocommit was enabled and there was no BEGIN, the SQL layer will automatically commit at the end of each statement.
- In InnoDB, commit updates the undo log information in a mini-transaction:
  1. Update undo log directory (rollback segments) and undo log header pages.
  2. Commit the mini-transaction (write to the redo log buffer).
  3. Obtain the last written LSN (redo log sequence number) in the buffer.
  4. Optionally, ensure that all redo log buffer is written at least up to the LSN.

Alternate step 4a: Rollback

- Rollback is expensive because
  - All changes are done in reverse (jumping read pointers)
  - Reading undo log or index pages that were evicted from the buffer pool to disk
  - Secondary index leaf records might need removal (must look up primary key records)
- For each undo log record from the end to start (or savepoint), the log is applied in reverse:
  - For UPDATE: the old version of the data is read and an update vector calculated
- The same process as the update step is followed but reversed
  - 1 mini-transaction per clustered index
  - 2 mini-transactions per secondary index
- The undo log pages are freed in a maintenance operation called *Truncate undo log tail*.

Step 4b: Cleaning up

After *User* `COMMIT` or `ROLLBACK` is done, there are some clean-up steps:

- Transactional row locks can be released and waiting threads woken up.
- InnoDB will also wake up any other transactions that waited for the locks.
- If the binlog is enabled, a commit event will be synchronously written to it.
  - An internal `XA PREPARE` transition must have been persisted in the InnoDB redo log earlier.
  - Only the internal `XA PREPARE` but not the `XA COMMIT` needs to be flushed to redo log.
- Finally, the SQL layer will release metadata locks on the table name – the table can now be `ALTER`ed again

Mini-Transaction Layer

The InnoDB Mini-Transaction Layer

Each layer provides service to the upper layers, relying on the service provided by the lower layer, adding some encapsulation or refinement:

- File system (or the layers below): Block address translation
  - Wear leveling, bad block avoidance
  - Deduplication
  - Redundancy or replication for fault tolerance or HA
- The InnoDB mini-transaction layer depends on a write-ahead *redo log*.
  - Provides an AcID service of modifying a number of persistent pages.
    - Example: Inserting a record into a B-tree, with a series of page splits.
  - An atomic mini-transaction becomes durable when its log is fully written to the redo log.
  - Recovery: Any redo log records after the latest checkpoint will be parsed and applied to the buffer pool copies of referenced persistent data file pages.

Mini-Transactions and Page Locking

A mini-transaction is **not** a user transaction. It is a short operation comprising:

- A list of index, tablespace or page latches that have been acquired.
- A list of modifications to pages.

There is **no rollback** for mini-transactions.

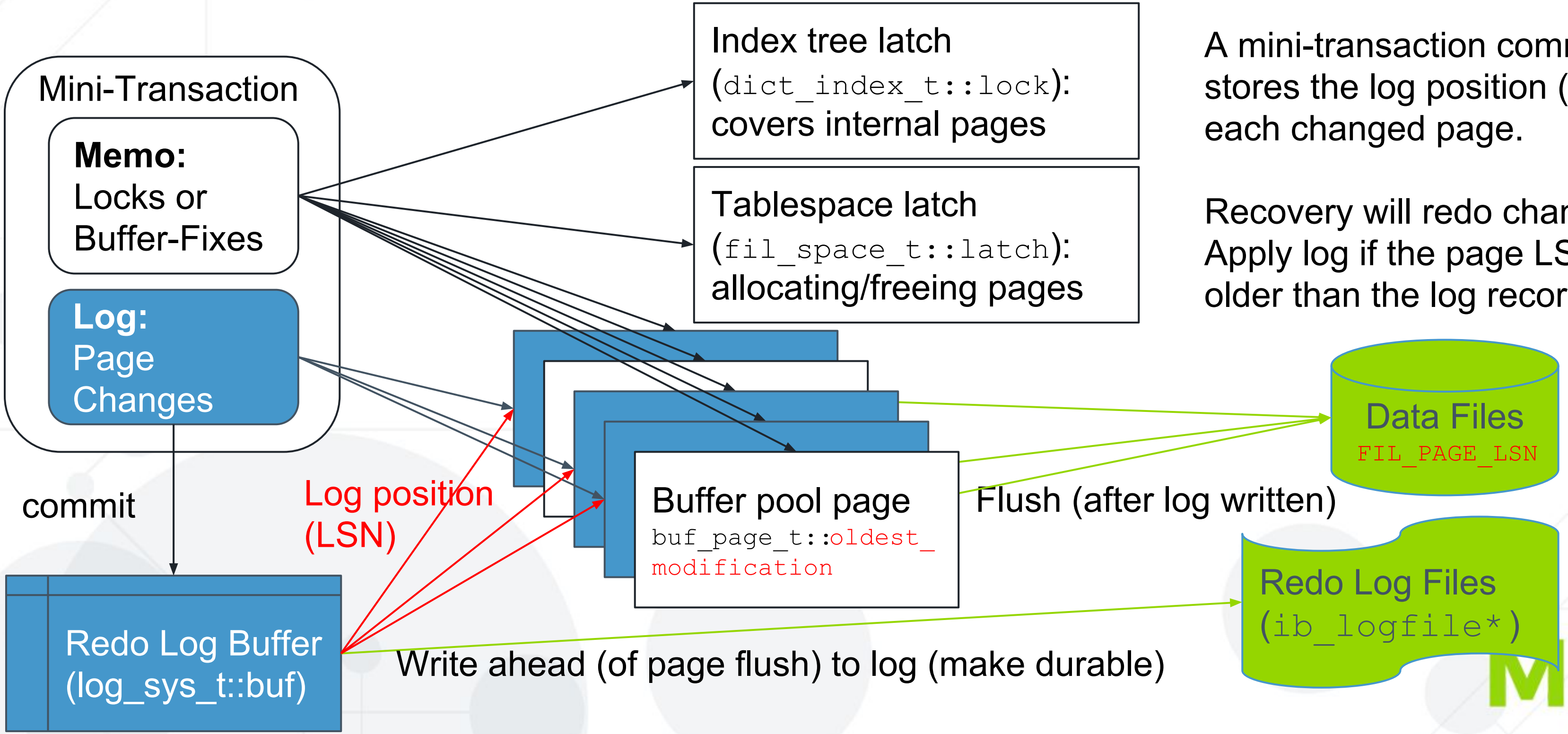
Latches of unmodified pages can be released any time. Commit will:

1. Append the log (if anything was modified) to the redo log buffer
2. Release all locks

Index page latch acquisition must avoid deadlocks (server hangs):

- Backward scans are not possible. Only one index update per mini-transaction.
- Forward scan: acquire next page latch, release previous page (if not changed)

Mini-Transactions: RW-Locks and Redo Logs



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.

Mini-transactions and Durability

Depending on when the redo log buffer was written to the redo log files, recovery may miss some latest mini-transaction commits.

When `innodb_flush_log_at_trx_commit=1`, the only mini-transaction that is synchronously persisted to disk is that of a *User transaction commit*. All earlier mini-transactions are also persisted.

Mini-transaction Write operations on Index Trees

In InnoDB, a table is a collection of indexes: PRIMARY KEY (clustered index, storing all materialized columns), and optional secondary indexes.

A mini-transaction can only **modify one index at a time**.

- *Metadata change:* Delete-mark (or unmark) a record
- *Insert:* optimistic (fits in leaf page) or pessimistic (allocate page+split+write)
    - Allocate operation modifies the page allocation bitmap page belonging to that page number
    - Tablespace extension occurs if needed (file extended, tablespace metadata change)
- *Delete (purge,rollback):* optimistic or pessimistic (merge+free page)
- *Index lookup:* Acquire index lock, dive from the root to the leaf page to edit

M|18 MariaDB

Read Mini-Transactions in Transactions

In InnoDB, a table is a collection of indexes: `PRIMARY KEY` (clustered index, storing all materialized columns), and optional secondary indexes.

A mini-transaction can read only 1 secondary index, the clustered index and its undo log records

- *Index lookup:* Look up a single record, dive from the root to the leaf page.
  - If this version is not to be seen, InnoDB will read the older version using `DB_ROLL_PTR`
- *Index range scan:* Index lookup followed by:
  - release locks except the leaf page
  - acquire next leaf page lock, release previous page lock, …
  - Typical use case: Filtering records for the read view, or for pushed-down `WHERE` clause

- *Encryption key rotation:* Write a dummy change to an encrypted page, to cause the page to be encrypted with a new key.

Flushing dirty pages

- This process runs asynchronously from the transactions
- Every second up to `innodb_io_capacity` iops will be used to write changes to data files
- Includes (potentially no longer needed) undo log pages and the contents of freed pages. To be improved: Collect garbage, do not write it!
- Data-at-rest encryption and page compression is done at this point
- Pages with oldest changes are written first
    - InnoDB keeps a list of dirty pages, sorted by the mini-transaction end LSN

Redo Log Checkpoint (Truncating Redo Log)

On startup, any redo log after the latest checkpoint will be read and applied.

- A redo log checkpoint logically truncates the start of the redo log, up to the LSN of the oldest dirty page.
- A clean shutdown ends with a log checkpoint.
- The InnoDB master thread periodically makes log checkpoints.
- When the redo log is full, a checkpoint will be initiated. Aggressive flushing can occur on any mini-transaction write.
    - Until enough space is freed up, the server blocks any other write
    - This is very bad for performance, a bigger `innodb_log_file_size` would help

One More Layer: the Doublewrite Buffer

What if the server was killed in the middle of a page flush?

- On Linux, killing a process may result in a partial write (usually 4096*n* bytes)
- Power outage? Disconnected cable? Cheating `fsync()`?
- On Windows with innodb_page_size=4k and matching sector size, no problem.

Page writes first go to the doublewrite buffer (128 pages in ibdata1).

- If startup finds a corrupted page, it will try doublewrite buffer recovery.
- If it fails, then we are out of luck, because redo log records (usually) do not contain the full page image.

Bugs and improvement potential: MDEV-11799, MDEV-12699, MDEV-12905

M|18 MariaDB

Crash Recovery

On startup, InnoDB performs some recovery steps:

- Read and apply all redo log since the latest redo log checkpoint.
  - While doing this, restore any half-written pages from the doublewrite buffer.
  - A clean shutdown ends with a log checkpoint, so the log would be empty.
- Resurrect non-committed transactions from the undo logs.
  - Tables referred to by the undo log records are opened.
  - This also resurrects implicit locks on all modified records, and table locks to block DDL.
  - `XA PREPARE` transactions will remain until explicit `XA COMMIT` or `XA ROLLBACK`.
- Initiate background `ROLLBACK` of active transactions.
  - Until this is completed, the locks may block some operations from new operations.
  - DBA hack: If you intend to `DROP TABLE` right after restart, delete the `.ibd` file upfront

Transaction Isolation Levels and Multi-Version Concurrency Control

Read Views and Isolation Levels

The lowest transaction isolation level is **READ UNCOMMITTED**.

- Returns the newest data from the indexes, “raw” mini-transaction view.
- Indexes may appear inconsistent both with each other and internally: an `UPDATE` of a key may be observed as a `DELETE`, with no `INSERT` observed.

REPEATABLE READ uses non-locking *read views*:

- Created on the first read of a record from an InnoDB table, or
- Explicitly by `START TRANSACTION WITH CONSISTENT SNAPSHOT`

READ COMMITTED is similar to REPEATABLE READ, but the read view is

- Created at the start of each statement, on the first read of an InnoDB record

Locks and Isolation Levels

The highest transaction isolation level **SERIALIZABLE** implies

`SELECT…`**`LOCK IN SHARE MODE`**

- Only a committed record can be locked. It is always the latest version.

- `DELETE` and `UPDATE` will internally do `SELECT…FOR UPDATE`, acquiring explicit exclusive locks.
- `INSERT` initially acquires an implicit lock, identified by `DB_TRX_ID` pointing to a non-committed transaction.

Read Views, Implicit Locks, and the Undo Log

InnoDB may have to read undo logs for the purposes of:

- Multi-versioned (non-locking) reads based on a read view:
  - Get the appropriate version of a record, or
  - “Unsee” a record that was inserted in the future or whose deletion is visible.
- Determining if a secondary index record is locked by an active transaction.
  - The PRIMARY KEY record is *implicitly locked* if `DB_TRX_ID` refers to an active transaction.
  - On conflict, the lock waiter will convert the implicit exclusive lock into an explicit one.
- Transaction rollback
  - Rollback to the start of statement, for example on duplicate key error
  - `ROLLBACK` [`TO SAVEPOINT`]
  - Explicit locks will be retained until the transaction is completed
  - Implicit locks are released one by one, for each rolled-back row change.

MVCC Read Views and the Undo Log

Multi-versioning concurrency control: provide non-locking reads from a virtual 'snapshot' that corresponds to a *read view*: The current *DB_TRX_ID* and a list of *DB_TRX_IDs* that are not committed *at that time*.

The read view contains:

- Any changes made by the current transaction.
- Any records that were committed when the read view was created.
- No changes of transactions that were started or committed after the read view was created.

"Too new" `INSERT` can be simply ignored. `DELETE` and `UPDATE` make the previous version available by pointing to the undo log record.

M|18 MariaDB

MVCC Read Views in the `PRIMARY` Index

The hidden `PRIMARY` index fields `DB_TRX_ID`, `DB_ROLL_PTR` and undo log records constitute a singly-linked list, from the newest version to the oldest.

A non-locking read iterates this list in a loop, starting from the newest version:

- If `DB_TRX_ID` is in the read view:
  Return the record, or skip it if delete-marked.
- If `DB_TRX_ID` is in the future and `DB_ROLL_PTR` carries the “insert” flag:
  Skip the record (the oldest version was inserted in the future).
- Else: Find the undo log record by `DB_ROLL_PTR`, construct the previous version, go to next iteration.

MVCC Read Views and Secondary Indexes

Secondary indexes only contain a `PAGE_MAX_TRX_ID`.

- Secondary indexes may contain multiple (*indexed_col*, *pk*) records pointing to a single *pk*, one for each value of *indexed_col*.
- All versions except at most one must be delete-marked.
- If a secondary index record is delete-marked, MVCC must look up *pk* in the `PRIMARY` index and attempt to construct a version that matches *indexed_col*, to determine if (*indexed_col*, *pk*) exists in the read view.
- If the `PAGE_MAX_TRX_ID` is too new, for each record in the secondary index leaf page, we must look up *pk*.
- For old read views, secondary index scan can be very expensive!

Purging Old History

Purging Old History

```
UPDATE talk SET attendees = 25 WHERE conference="M|18"
    AND name="Deep Dive";
```

- If there were any secondary indexes that depend on the updated columns, we delete-marked the records that contained the old values, before inserting attendees=25. Those records should be removed eventually.
- In the primary key index, in MariaDB 10.3 we would set `DB_TRX_ID=0` after the history is no longer needed, to speed up future MVCC and locking checks.
- Also, the undo log pages are eventually freed for reuse, in a mini-transaction *Truncate undo log head*.

Purge Lag and Long-Running Transactions

The **purge threads** can start removing history as soon as it is no longer visible to any active *read view*.

- **READ COMMITTED** switches read views at the start of each statement.
- **REPEATABLE READ** keeps the read view until commit (or full rollback).

Undo log is by default stored in the InnoDB system tablespace (ibdata1).

- Because of purge lag, this file could grow a lot. InnoDB files never shrink!
- Open read views prevent purge from running.
- Purge lag (long `History list length` in `SHOW ENGINE INNODB STATUS`) causes secondary indexes and undo logs to grow, and slows down operations.

Secondary Indexes and the Purge Lag

In the worst case, MVCC and implicit lock checks will require a clustered index lookup for each secondary index record, followed by undo log lookups.

- Updates of indexed columns make this **very expensive**: $O(versions \cdot rows^2)$.
- Ensure that purge can remove old history: **Avoid long-lived read views.**
- **Avoid secondary index scans in long-lived read views.**
- Watch the `History list length` in `SHOW ENGINE INNODB STATUS`!

Thank you!

To be continued in the other Deep Dive:
InnoDB Transactions and Replication