L-Store Milestone 3

Alana Rufer, Eseosa Omorogieva, Nina Gopal,
Riddhi Barbhaiya, Kushaal Rao

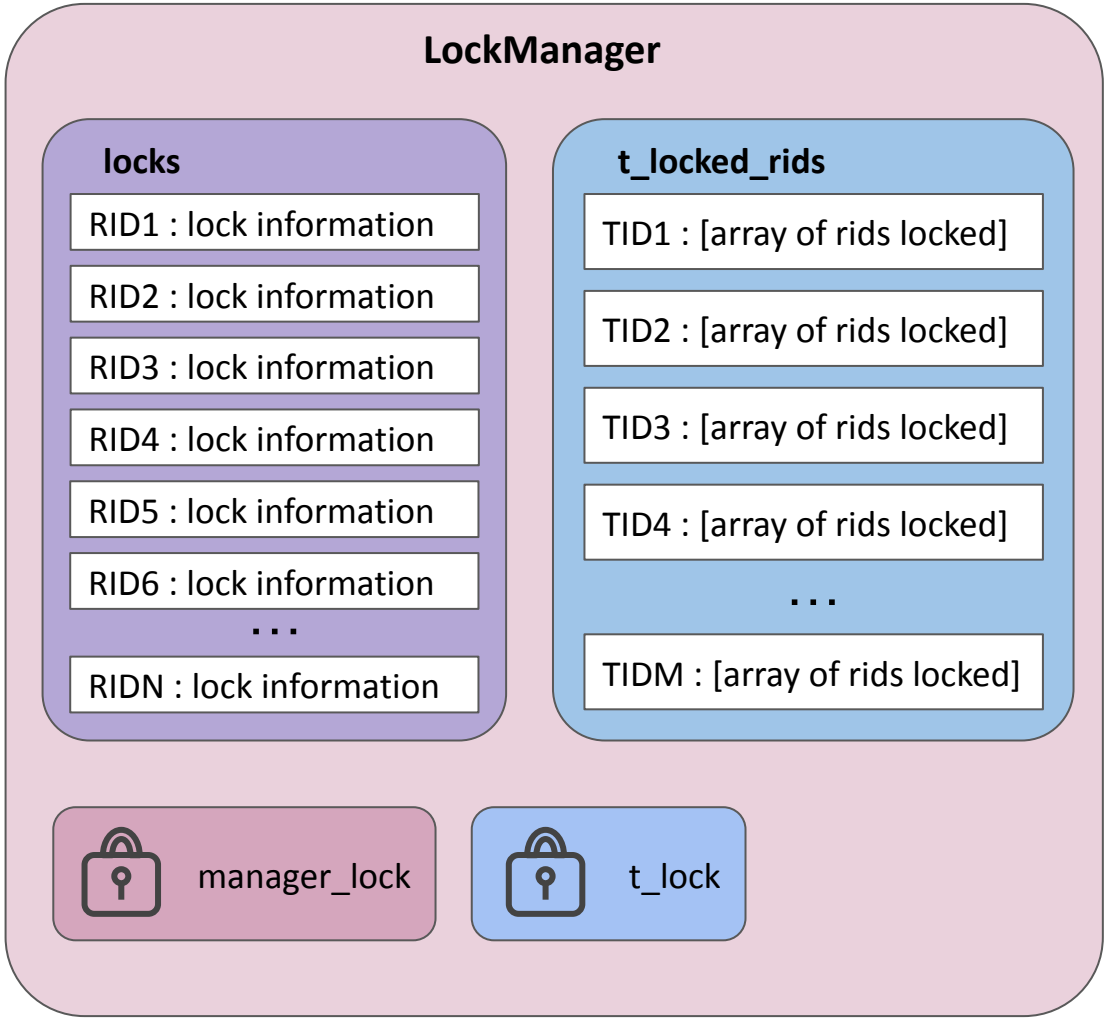
Lock Manager

LockManager

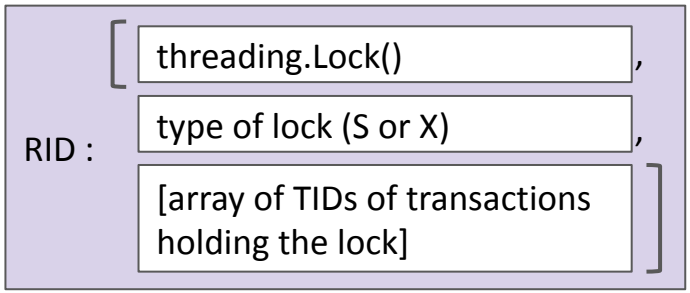
Manages shared and exclusive locks used by 2PL

Coordinates release of all locks held by a transaction

Locks initialized during db.open() to reduce performance overhead of creating locks



Locking logic (for records)



If requested lock is not available, transaction immediately aborts (NoWait)

Lock() object provides thread-safety for lock information.

Actual holding of lock stored in type of lock and array of transaction TIDs

Locks currently held on specified record

Lock requested by Xact	None	S lock (same Xact)	X lock (same Xact)	S lock (different Xact)	X lock (different Xact)
S lock	acquire	no action	no action	acquire	abort
X lock	acquire	upgrade (if no other Xacts holding the lock) else abort	no action	abort	abort

Latching

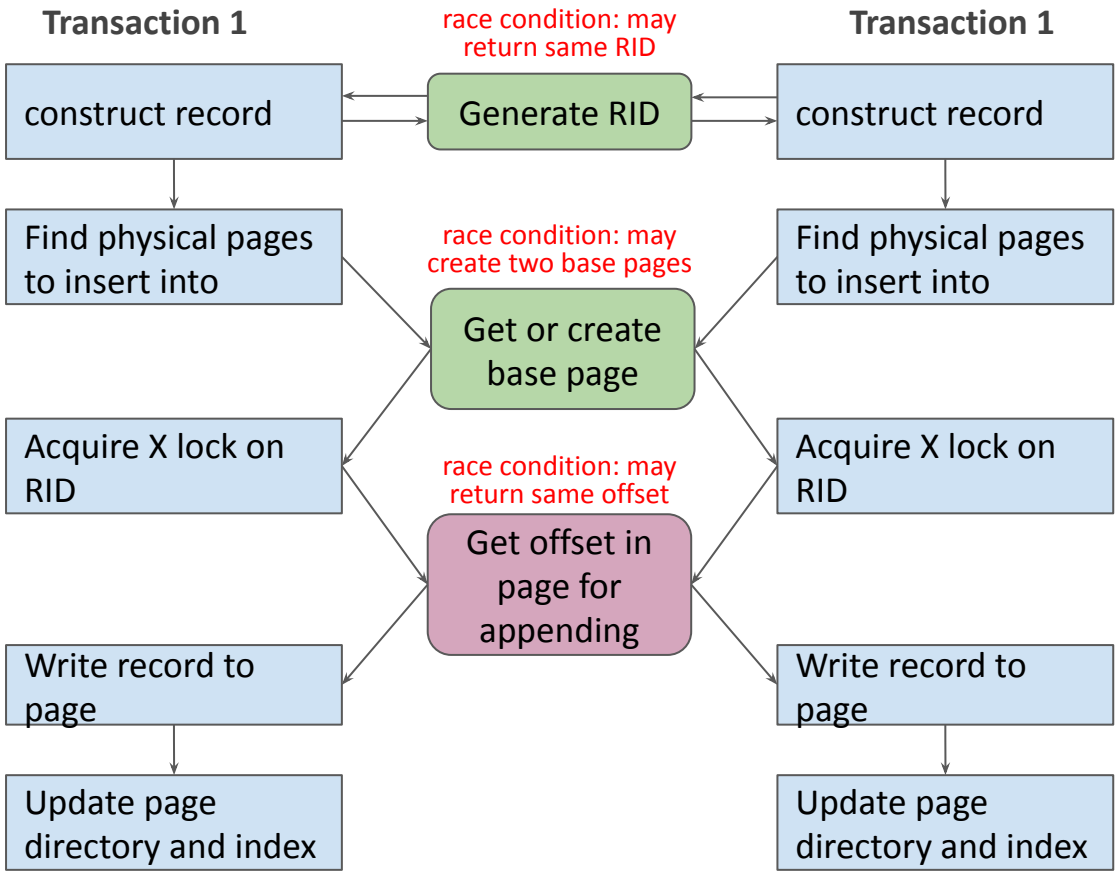
Latching overview

Conflicts from accessing or modifying shared data

threading.Lock() for basic locking

Readers-Writer Lock: increase concurrency by allowing multiple readers when possible

Ex: two transactions insert at the same time (without latching):



Preventing Conflicts in Shared Data Structures

RID generation

page_directory:
accessing/
modifying entries

LogicalPage:
accessing/
modifying pageids

Index:
accessing/
modifying entries

Merge:
starting merge,
adding to merge list

db & transaction & create_index:
Readers-Writer Lock (writer-preferring)

- "writers": create_index, db.close()
- "readers": transactions

Guarantee success of create_index()
Wait for transactions to finish before db.close()

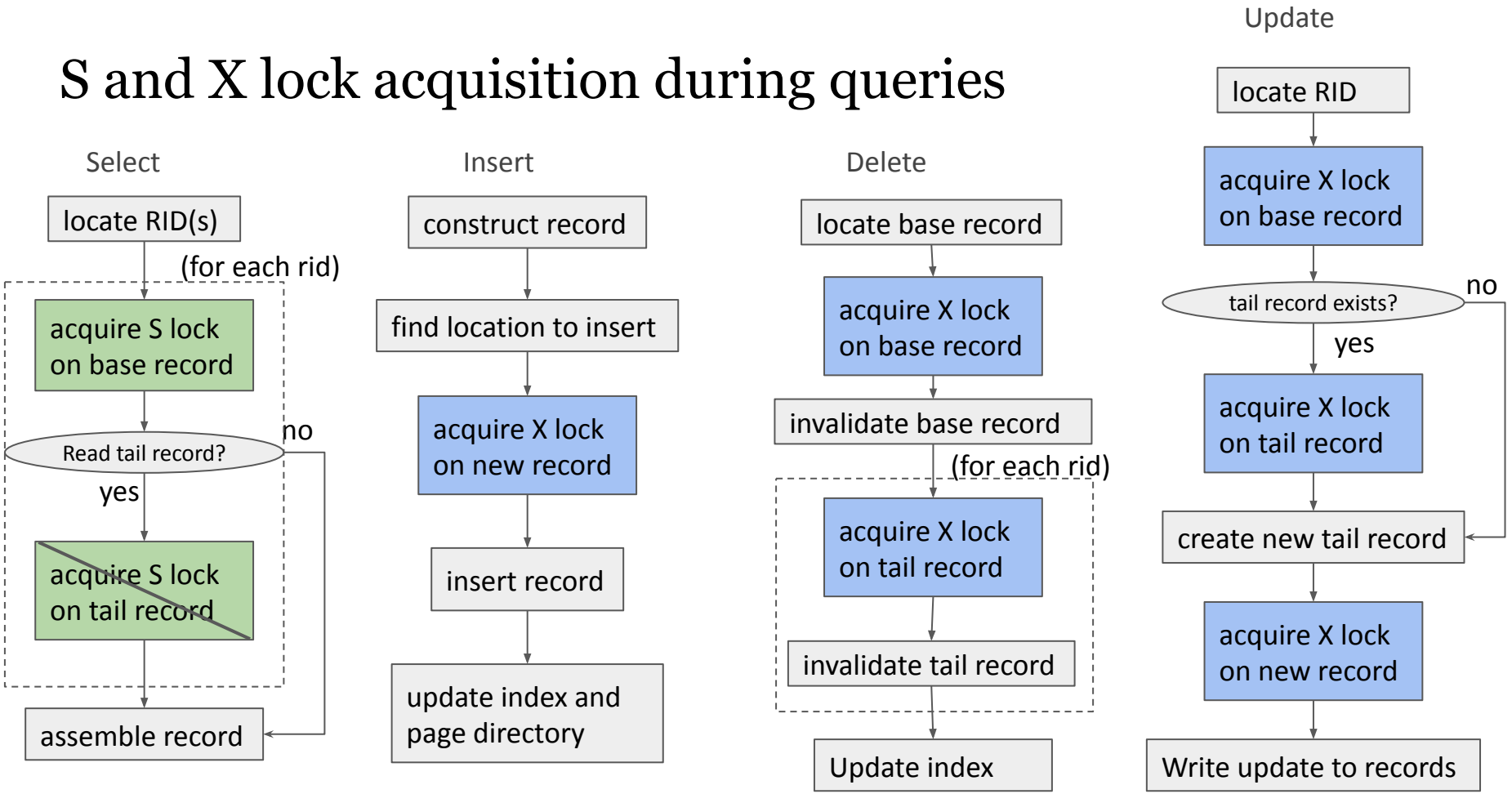
Insertion of record:
Prevent conflict on location in page, or double-creation of new page

Separate locks for base page and each active tail page

Bufferpool:
file pointer seek + r/w

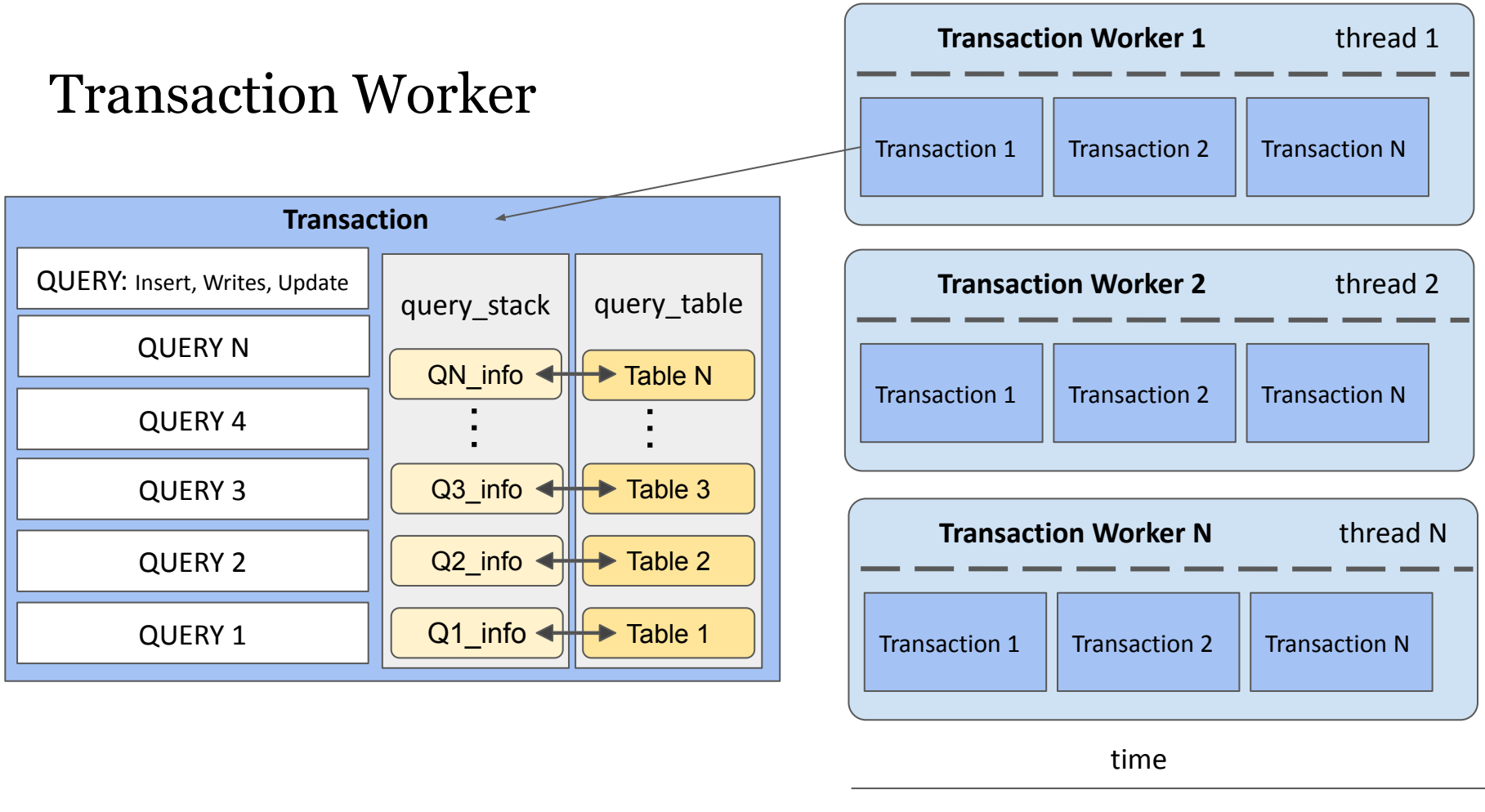
accessing sensitive operations and data structures

S and X lock acquisition during queries



Transaction & Transaction Worker

Transaction Worker



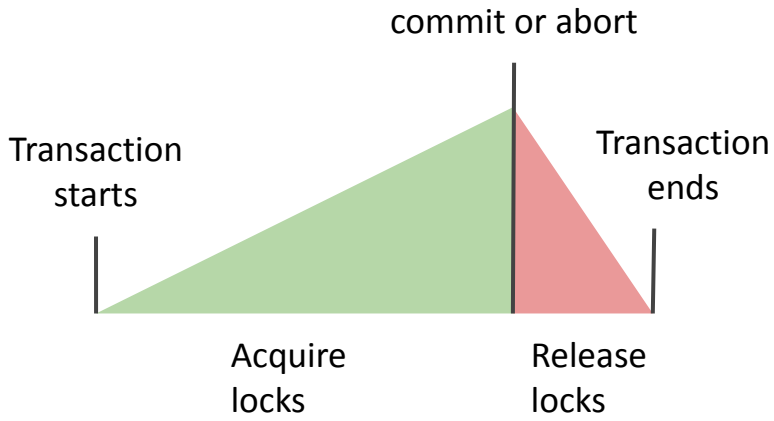
Atomicity & Isolation

Serializability via **2PL**

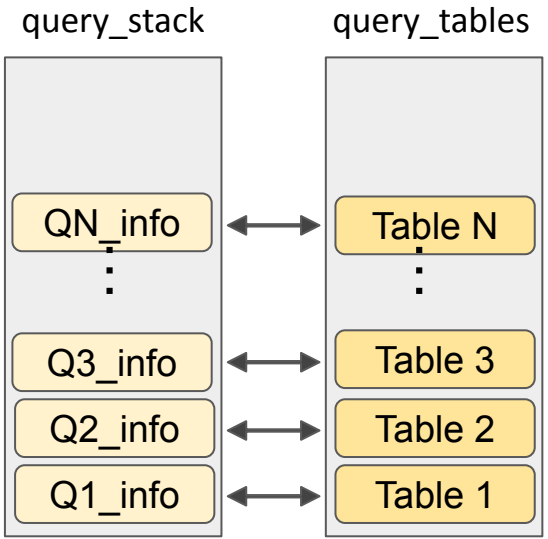
If transaction is successful it is committed to the database

On failure to acquire a lock, transaction aborts immediately. All changes are rolled back

After transaction commits or aborts, all record locks held by the transaction are released (transaction atomicity)



Aborting Transactions



Store info about completed transactions and their corresponding tables in two stacks

During the abort, we pop the stacks and undo each query

Release all rid locks during commit and abort

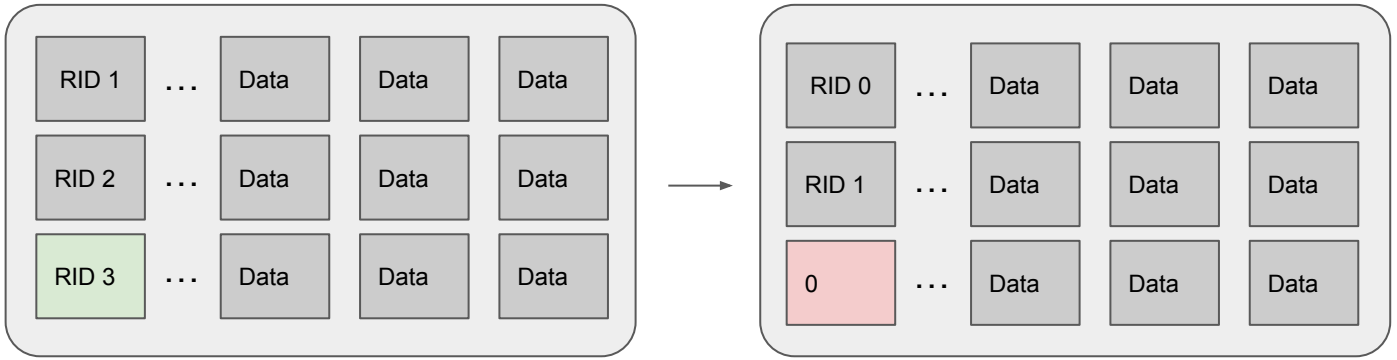
Query Info

Insert: (Success_value, “I” , inserted_rid, columns)

Delete: (Success_value, “D” , invalidated_rids, fields)

Update: (Success_value, “U” , only_locks, rid_to_update, last_update_rid, new_rid, old_schema_encoding, old_values, new_values, columns_modified)

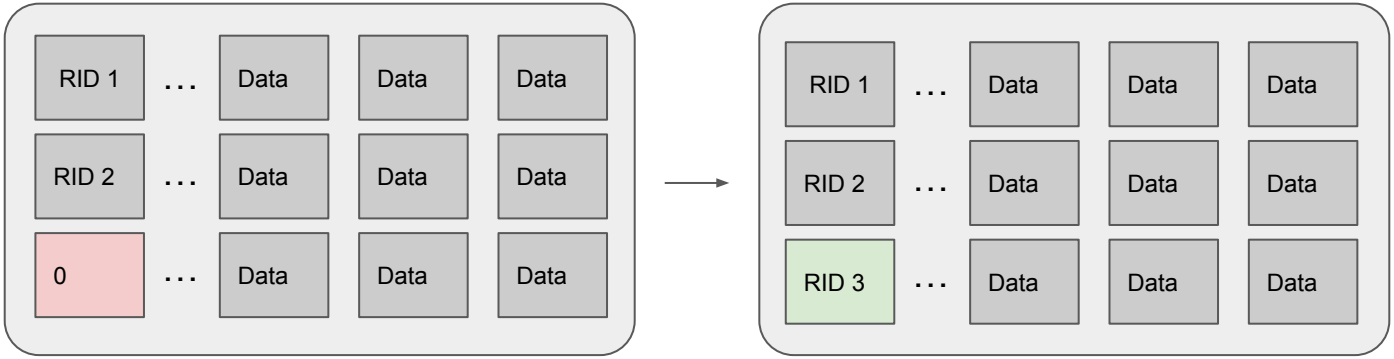
insert_undo



Mark RID of inserted record as invalid

Remove the RID from the index on the columns if exists

delete_undo



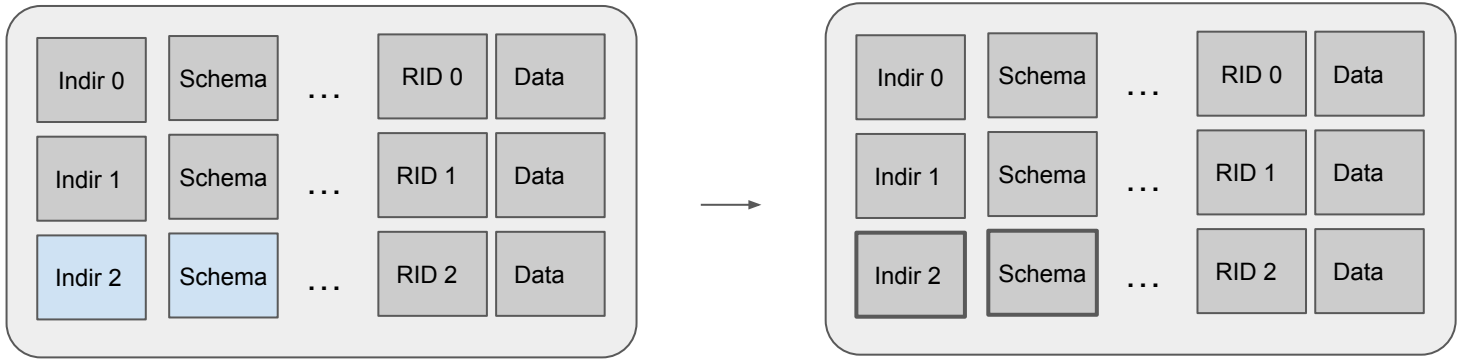
Restore the invalid RID of the base record

Add the RID to the index on the columns if exists

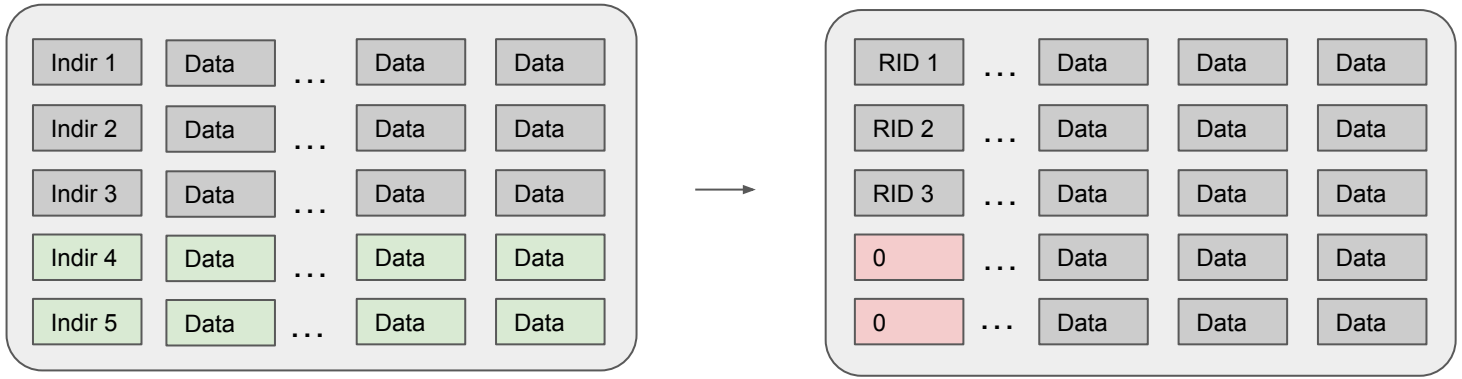
update_undo

Update: (Success_value, “U” , only_locks, rid_to_update, last_update_rid, new_rid, old_schema_encoding, old_values, new_values, columns_modified)

Base Page



Tail Page

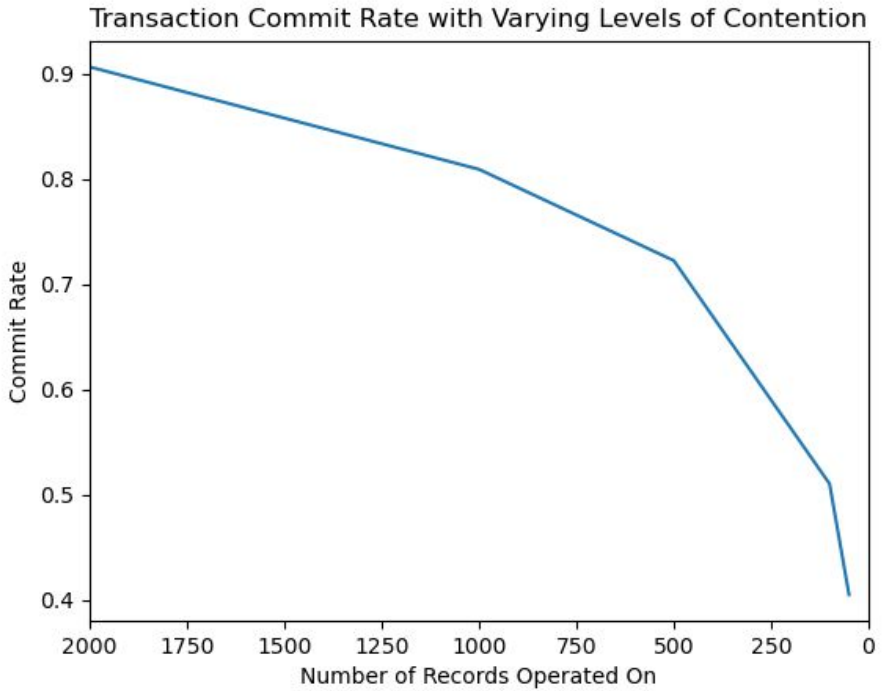


Performance

Commit Rates with Varying Contention

Workload

- 2 threads
- 25 transactions per thread
- update 1000 times (randomly choose the key to update)
- Vary the number of records in the db
  - Less records- more contention between transactions



Hardware: Dual-Core Intel Core i7, 2.5GHz, 16GB, 4 MB L3 Cache

Q&A