



INNODB[®]

Transactional Storage for MySQL
FAST. RELIABLE. PROVEN.

Concurrency Control: How It Really Works

MySQL Conference, April 2009

Heikki Tuuri, CEO, Innobase Oy

INNOBASE

Topics

- Why Do We Need Concurrency Control
- ACID Transactions
- Consistent Read
- Shared And Exclusive Locks
- Auto-Increment Locking
- Row Locks, Gap Locks
- Deadlock Detection and Resolution

Why Do We Need Concurrency Control?

- A 'transaction' transfers your database from one legal state to another
- A typical example is a money transfer from account A to account B
- You need to lock data to make sure that relevant parts of your database do not change when you process the transaction
- Also, you want your queries to see the database in a legal state: a 'consistent read' is for that

InnoDB Transaction Model and Locking

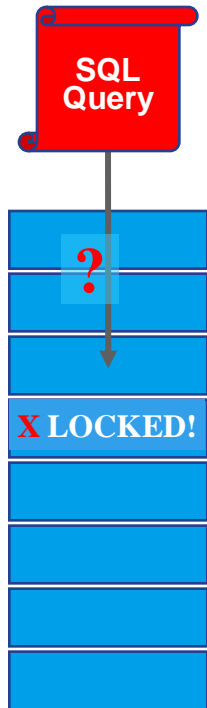
- Combines the best properties of multi-versioning with traditional two-phase locking
- InnoDB locks at the row level
- InnoDB runs queries as non-locking consistent reads by default, in the style of Oracle

Typically, users can lock every row in the database, or any random subset of the rows, without InnoDB running out of memory

InnoDB Transactions

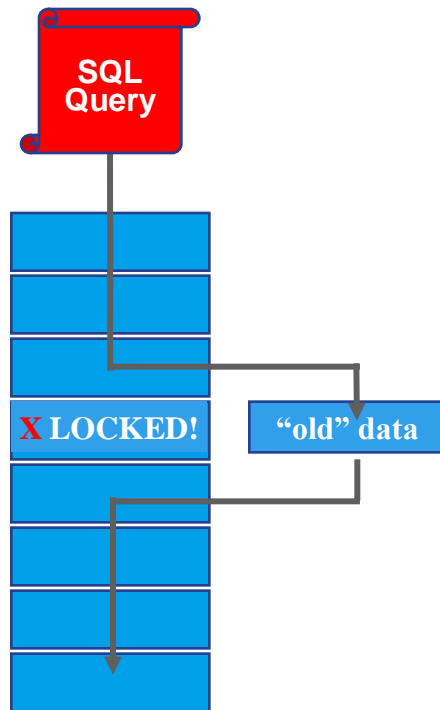
- **A**tomic - all changes are either committed as a group, or all are rolled back as a group
- **C**onsistent - transactions operate on a consistent view of the data, leaving the data in a consistent state (by transaction's end)
- **I**solated - each transaction “thinks” it is running by itself - effects of other transactions are invisible until it commits
- **D**urable - once committed, all changes persist, even if there are system failures

InnoDB Transactions & Locking



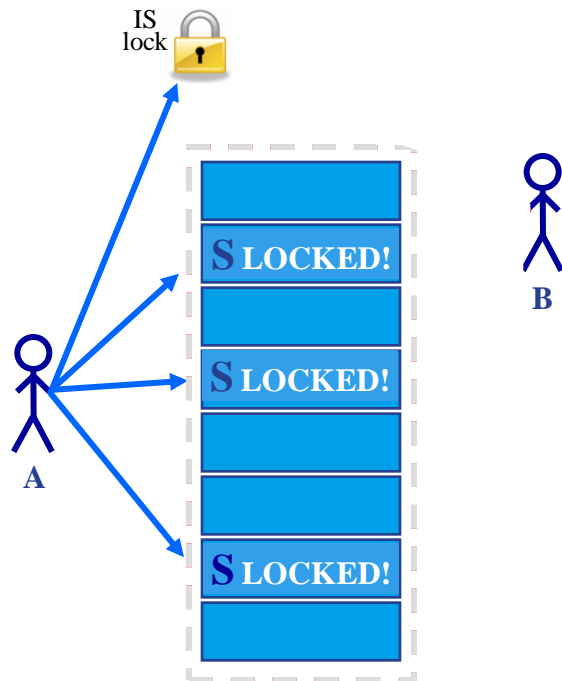
- Full transaction support
 - atomicity, consistency, isolation, durability
 - SQL-standard isolation levels
 - automatic deadlock detection
- Unlimited row-level locking
- Multi-version read-consistency

InnoDB Consistent Reads



- Queries see a snapshot of the data consistent with the other data they read
- By default, InnoDB uses “consistent read” for queries like this
`SELECT a FROM t WHERE b = 2;`
- Normal undo is used to generate consistent data for the query to see
 - No overhead: undo info is required to rollback uncommitted transactions
- No need to set locks, as history cannot change

Shared and Exclusive Locks



Q: If user A has shared row locks in table T, how does InnoDB know to not let user B set an exclusive X lock on table T?

A: 'Intention locks'. Before setting a shared lock on a row in t, user A sets an 'intention lock' IS on table t.

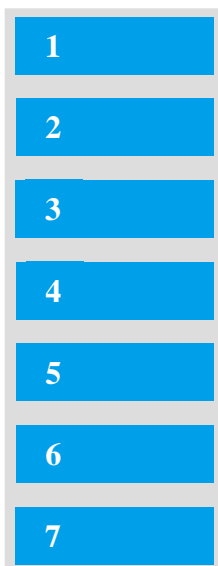
IS is compatible with S, but not with X. Thus, if a user has row share locks, no other user can lock the table in X mode

Similarly, before setting an exclusive X lock on a row, a user must set an IX lock on the table.

- IX is not compatible with S lock on T
- if a user has IX lock on table T, no other user can take S lock on T
- if a user has an S lock on table T, no other user can take X locks on rows in T

Auto-Increment Locking

- Statement-based replication require auto-increment values to be deterministic
- InnoDB uses a table-level 'auto-increment lock'
 - Table-level lock occurs at time of INSERT
 - Lock is released at statement end, not transaction end
- This table-level auto-increment lock can become a bottleneck in high-concurrency (> 10 threads) INSERT workloads
- Row-based replication allows InnoDB to release the auto-increment lock early => better concurrency



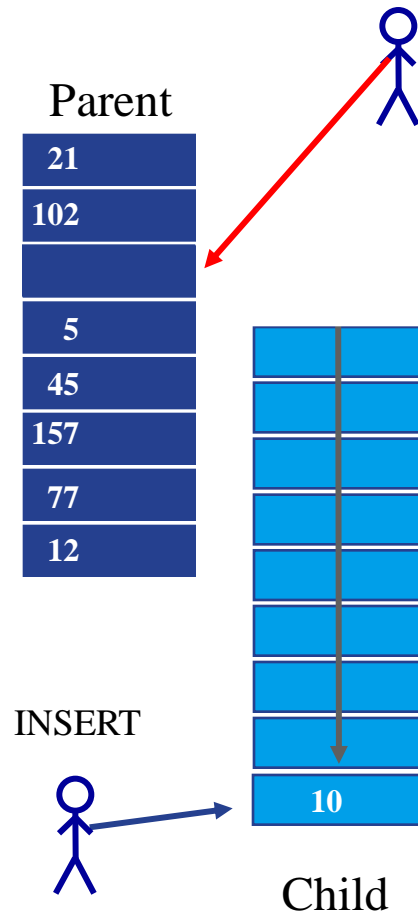
Auto-Increment Locking (2)

- As of MySQL 5.1.22, InnoDB can avoid using the table-level AUTOINC lock for a class of INSERT statements where the number of rows is known in advance
- With `innodb_autoinc_lock_mode = 2` (“interleaved” lock mode), no “INSERT-like” statements use the table-level AUTOINC lock
 - multiple statements can execute at the same time
 - the fastest and most scalable lock mode
 - not safe with statement-based replication or recovery scenarios when SQL statements are replayed from the binlog
 - auto-increment values are guaranteed to be unique and monotonically increasing across all concurrently executing “INSERT-like” statements
 - However, because multiple statements can be generating numbers at the same time (“interleaved”), the values generated for any given statement may not be consecutive

USE ROW-BASED REPLICATION WITH VALUE 2!

INNOBASE

Phantoms vs. Consistency



PHANTOM: A row that appears in a second query that was not in the first

Example: foreign key check in application code

Check that there are no children with parent id=10:

```
SELECT * FROM child  
WHERE parent_id = 10 FOR UPDATE;
```

```
DELETE FROM parent WHERE id = 10;
```

- If the SELECT returns 0 rows, then the user thinks he can delete the parent
- But before first user COMMITs, another user inserts a child with parent_id = 10 ...

➔ INCONSISTENCY!

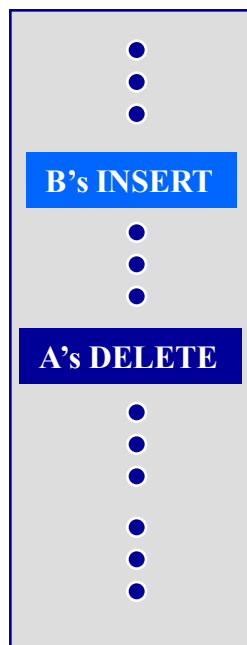
Statement-Based Replication Relies on No Phantoms



```
User A
BEGIN;
DELETE FROM t
WHERE a = 10;
COMMIT;
```



```
User B
BEGIN;
INSERT INTO t
VALUES (10);
COMMIT;
```



MySQL
binlog

1. User A deletes a row
2. Before A commits, user B inserts the *same* row
3. User B commits before A
4. User A commits
5. The MySQL binlog contain B's transaction before A's

We do not know if A deleted the row that B inserted!

 **slave may get out-of-sync with the master!**

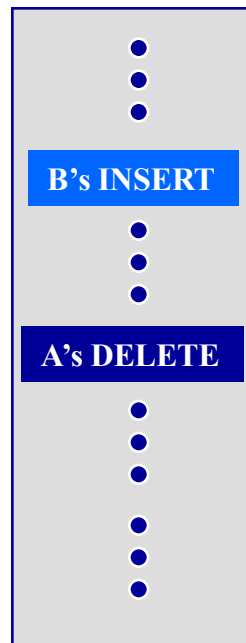
Row-Based Replication Needs Less Locking



```
User A
BEGIN;
DELETE FROM t
WHERE a = 10;
COMMIT;
```



```
User B
BEGIN;
INSERT INTO t
VALUES (10);
COMMIT;
```



MySQL
binlog

1. User A deletes a row
2. Before A commits, user B inserts the *same* row
3. User B commits before A
4. User A commits
5. The MySQL binlog contain B's transaction before A's

Row-based binlog contains information of each individual row that was deleted or inserted => phantoms are no longer a problem!

InnoDB Avoids Phantoms Through 'Gap Locking'

- Every SELECT, UPDATE, DELETE in InnoDB uses an index to find the rows to return or operate on
 - ➔ the index is searched, or scanned
- To avoid phantoms, we lock not only the index records we scan, but also the 'gaps' between them
 - ➔ No other user can insert new records in the gaps



If the query scans the rows between Heikki and Ken, we also lock the 'gap' between those records, so that other users cannot insert 'Jeffrey' in the gap

Types of Gap Locking in InnoDB

**Next-key
lock**

locks the key & the gap before the key

**Gap
lock**

locks just the gap before the key

**Record-only
lock**

locks just the key and not the gap

**Insert-
intention
gap lock**

held when waiting to insert into a gap

InnoDB minimizes gap locking by using
record-only locks in UNIQUE searches

```
UPDATE t SET a = a + 1 WHERE primary_key_value = 100;
```

InnoDB's Gap Locks are 'Purely Inhibitive'

InnoDB allows several users to have conflicting lock modes on the same gap. User A and user B are BOTH allowed to hold an X-lock on the same gap!

Why is this? The InnoDB purge (garbage collection) may remove a deleted record from an index. Two gaps then merge. The new, 'bigger' gap inherits the locks from the the two gaps, as well as the possible lock that was on the deleted record.

'gap' record r

Q: If a user holds an exclusive next-key lock on record r, is it guaranteed that he can always insert to the gap before r?

A: No. Another user may also have a gap lock, or a waiting next-key lock request on r! InnoDB's gap locks are 'purely inhibitive': they block others from inserting, but do not give the lock owner any privilege to insert to the gap.

Where Does InnoDB Store Its Locks?

- InnoDB stores table locks in memory
- INSERTs, UPDATEs, and DELETEs update the transaction id field in the row
- Row lock information in the transaction id field is called an 'implicit row lock' in InnoDB; that is, it is stored on disk
- 'Explicit row locks' are stored in bitmaps in memory
- SELECT ... FOR UPDATE sets explicit row locks

Which Index Records Can Carry Locks in InnoDB?

- Both clustered index (PRIMARY KEY index) and secondary index records can carry locks.
- An ordinary, existing, index record can, of course, carry a lock.
- A delete-marked index record can carry a lock.
- The 'supremum' pseudo-record on each InnoDB index page can carry a lock.
- The 'infimum' pseudo-record on each InnoDB index page cannot carry a lock.

Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

SERIALIZABLE

- All plain SELECTs execute as if they used LOCK IN SHARE MODE
- No 'consistent' reads; all SELECTs return the very latest state of the database
- Downside: lots of locking, lots of deadlocks.

REPEATABLE READ

- All SELECTs after the 1st consistent read SELECT in a transaction use the same “snapshot”
- UPDATE, DELETE use next-key locking
- This is the default level

INNOBASE

Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

**READ
COMMITTED**

- Each SELECT uses its own “snapshot”
- Data is “up to date”, but multiple SELECTs may be inconsistent with one another
- In V5.1, most gap-locking is removed w/ this level, but you **MUST** use row-based logging/replication
- Fewer gap locks mean fewer deadlocks
- UNIQUE KEY checks in secondary indexes and some FOREIGN KEY checks still need to set gap locks
 - Gaps must be locked to prevent inserting child rows after parent row is deleted
- Many users will move to this isolation level \geq V5.1
- Use `innodb_locks_unsafe_for_binlog` to remove gap locking in MySQL-5.0 and earlier

Transaction Isolation Levels

```
SET {SESSION | GLOBAL}  
TRANSACTION ISOLATION LEVEL <level>;
```

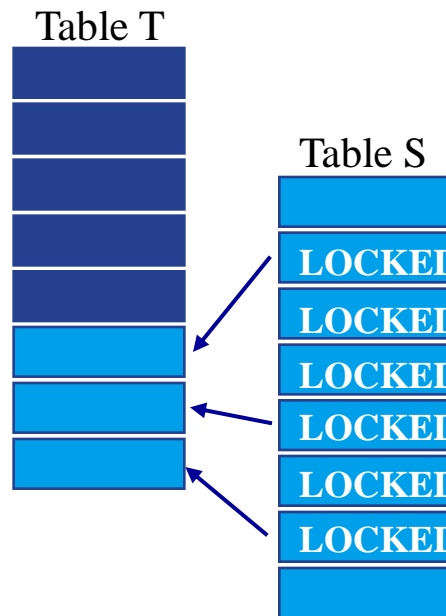
**READ
UNCOMMITTED**

- SELECTS will see data changes made by other users, but not committed
- **NOT RECOMMENDED**: no guarantee of data consistency

INSERTs w/ a SELECT Subquery



```
INSERT INTO t  
SELECT ...  
FROM s ...
```



- Using REPEATABLE READ (the default), InnoDB locks all records it scans in table S with share locks
- This is to prevent phantoms, which would break MySQL statement-based replication
- **Problem:** this type of query is often used in report generation
- **Fix:** use MySQL V5.1 and READ COMMITTED
 - Then table S is read using a consistent, non-locking read
 - Note: if the result in t is used in replication, then you **MUST** use row-based replication.

Locks Set by Different SQL Statements

- `SELECT ... FROM` is a consistent read, reading a snapshot of the database and setting no locks unless the transaction isolation level is set to `SERIALIZABLE`. For `SERIALIZABLE` level, the search sets shared next-key locks on the index records it encounters.
- `SELECT ... FROM ... LOCK IN SHARE MODE` sets shared next-key locks on all index records the search encounters.
- `SELECT ... FROM ... FOR UPDATE` sets exclusive next-key locks on all index records the search encounters and also on the corresponding clustered index records if a secondary index is used in the search.
- `UPDATE ... WHERE ...` sets an exclusive next-key lock on every record the search encounters.
- `DELETE FROM ... WHERE ...` sets an exclusive next-key lock on every record the search encounters.

Locks Set by Different SQL Statements (2)

- `INSERT INTO ... VALUES (...)` sets an exclusive lock on the inserted row. This lock is an index record lock without a gap lock (that is, it is not a next-key lock) and does not prevent other sessions from inserting into the gap before the inserted row. If a duplicate-key error occurs, a shared lock on the duplicate index record is set.
- `REPLACE` is done like an `INSERT` if there is no collision on a unique key. Otherwise, an exclusive next-key lock is placed on the row that must be updated.
- `INSERT INTO T SELECT ... FROM S WHERE ...` sets an exclusive non-next-key lock on each row inserted into T. InnoDB sets shared next-key locks on rows from S, unless [innodb_locks_unsafe_for_binlog](#) is enabled, in which case it does the search on S as a consistent read (no locks). InnoDB has to set locks in the former case: In roll-forward recovery from a backup, every SQL statement must be executed in exactly the same way it was done originally.

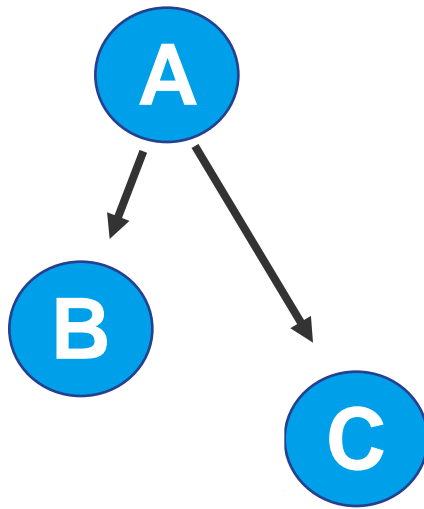
Locks Set by DDL Statements

- CREATE TABLE ...

Locks Set by Different SQL Statements (2)

- CREATE TABLE ... SELECT ... performs the SELECT with shared locks or as a consistent read, as in the previous INSERT ... SELECT item.
- If a FOREIGN KEY constraint is defined on a table, any insert, update, or delete that requires the constraint condition to be checked sets shared record-level locks on the records that it looks at to check the constraint. InnoDB also sets these locks in the case where the constraint fails.
- LOCK TABLES sets table locks, but it is the higher MySQL layer above the InnoDB layer that sets these locks. InnoDB is aware of table locks if `innodb_table_locks = 1` (the default) and `autocommit = 0`, and the MySQL layer above InnoDB knows about row-level locks.

Deadlock Detection & Rollback



*waits-for
graph*

- InnoDB automatically detects deadlocks if it detects a cycle in “waits-for” graph of transactions
- Given a deadlock, InnoDB chooses the transaction that modified the fewest rows as the victim, and rolls it back
- Note: InnoDB cannot detect deadlocks that span MySQL storage engines
 - Set `innodb_lock_wait_timeout` in `my.cnf`, to break deadlocks via timeout (default 50 sec)



Q
QUESTIONS
&
ANSWERS
A

INNOBASE

INNOBASE

developer of

INNODB[®]

and

INNODB[®] Hot Backup

ORACLE[®]

Innbase is an Oracle company

