Chapter 8
Lock Implementation
: InnoDB Locking

Mijin An
meeeeejin@gmail.com

VLDB Lab.

Locks in MySQL/InnoDB

- Multiple threads of execution access shared data structure

- InnoDB synchronizes these accesses with **mutexes** and **read/write locks**
  - A mutex to enforce exclusive-access locks to internal in-memory data structures
  - A rw-locks to enforce shared-access locks to internal in-memory data structures

Overview

- InnoDB has **eight** types of locks:
  1. Shared and Exclusive Locks
  2. Intention Locks
  3. Record Locks
  4. Gap Locks
  5. Next-key Locks
  6. Insert Intention Locks
  7. AUTO-INC Locks
  8. Predicate Locks for Spatial Indexes

1. Shared and Exclusive Locks

- A **row-level** lock

- **Shared locks (S locks)**
  - When reading data
  - Not mutually exclusive

- **Exclusive locks (X locks)**
  - When modifying data
  - Mutually exclusive

Type	S	X
S	Compatible	Conflict
X	Conflict	Conflict

2. Intention Locks (1)

- A **table-level** lock
- This lock indicates which type of lock a transaction requires later for a row in a table

- **Intentional Shared locks (IS locks)**
  - A transaction intends to set a S lock on individual row in a table

- **Intentional Exclusive locks (IX locks)**
  - A transaction intends to set a X lock on individual row in a table

- Intention locks only indicate intentions

2. Intention Locks (2)

- Compatible with each other:

Type	IS	iX
iS	Compatible	Compatible
iX	Compatible	Compatible

- Mutually exclusive with S/X locks:

Type	S	X
iS	Compatible	Conflict
iX	Conflict	Conflict

```
SELECT ... LOCK IN SHARE MODE;
SELECT ... FOR UPDATE;
```

3. Record Locks

- A **lock on an index record**

```
SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE;
```

- Above SQL prevents any other transaction from inserting, updating, or deleting rows where the value of `t.c1 = 10`

- Record locks always lock index records
  - If a table has no indexes, InnoDB uses a hidden clustered index for record locking

4. Gap Locks (1)

- A **lock on a gap between index records**
- Or A **lock on the gap before the first of after the last index record**

```
SELECT c1 FROM t WHERE c1 BETWEEN 10 and 20 FOR UPDATE;
```

- Above SQL prevents other transaction from inserting a value of 15 into column `t.c1`, because the gaps between all existing values in the range are locked

- The main purpose is to prevent other transactions from inserting data into the interval, leading to **non-repeatable** reading

4. Gap Locks (2)

- If the isolation level is reduced to **READ COMMITTED**, the gap lock is disabled

- Gap locks are **not needed** to search for a unique row using **a unique index**

```
SELECT * FROM child WHERE id = 100;
```

  - `id` column has a unique index, so it uses only an index-record lock for the row

5. Next-Key Locks

- A **record lock** on the index record **+** a **gap lock on the gap before the index record**

- Suppose that an index contains the values 10, 11, 13, and 20. The possible next-key locks for this index cover the following intervals:

```
(negative infinity, 10]
(10, 11]
(11, 13]
(13, 20]
(20, positive infinity)
```

- By default, InnoDB operates in **REPEATABLE READ** transaction isolation level:
  - Next-key locks are used for search and index scan to prevents **phantom rows**

6. Insert Intention Locks (1)

- A **type of gap lock set by `INSERT` operations prior to row insertion**
  - To improve insert concurrency

- This locks signal the intent to insert in such a way that multiple transactions inserting into the same index gap need not wait for each other if they are not inserting at the same position within the gap

6. Insert Intention Locks (2)

- Client A:

```
mysql> CREATE TABLE child (id int(11) NOT NULL, PRIMARY KEY(id)) ENGINE=InnoDB;
mysql> INSERT INTO child (id) values (90),(102);
mysql> START TRANSACTION;
mysql> SELECT * FROM child WHERE id > 100 FOR UPDATE;
+-----+
| id  |
+-----+
| 102 |
+-----+
```

- Client B:

```
mysql> START TRANSACTION;
mysql> INSERT INTO child (id) values (101);
```

7. AUTO-INC Locks

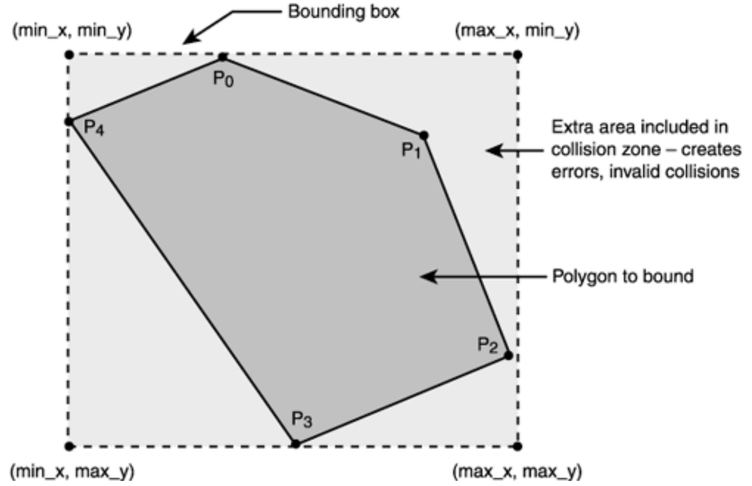
- A **table-level lock** taken by transactions inserting into tables with `AUTO_INCREMENT` columns

- If a transaction is inserting records into a table, any other transactions must wait, so that rows inserted by the first transaction receive consecutive primary key values

8. Predicate Locks for Spatial Indexes

- InnoDB supports `SPATIAL` indexing of columns containing spatial columns
  - Next-key locking does not work well to support **REPEATABLE READ** or **SERIALIZABLE** transaction isolation levels (not clear which is the “next” key)

- A `SPATIAL` index contains minimum bounding rectangle (MBR) values
  - InnoDB enforces consistent read on the index by setting a predicate lock on the MBR value used for a query
  - Other transactions cannot insert or modify a row that would match the query condition



Reference

[1] Jim Gray and Andreas Reuter, “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann, San Mateo, CA (1993)

[2] “22.3 InnoDB Mutex and Read/Write Lock Implementation”, MySQL Internals Manual, https://dev.mysql.com/doc/internals/en/innodb-mutex-rwlock-implementation.html

[3] “In-depth explanation of lock mechanism in MySQL”, Develop PAPER, https://developpaper.com/in-depth-explanation-of-lock-mechanism-in-mysql/

[4] “14.7.1 InnoDB Locking”, MySQL Internals Manual, https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html#innodb-shared-exclusive-locks