Chapter 6. Transactions and Locking

Using locks for transaction isolation is a pillar of SQL databases—but this is also an area that can cause a lot of confusion, especially for newcomers. Developers often think that locking is a database issue and belongs to the DBA realm. The DBAs, in turn, believe this is an application issue and consequently the responsibility of the developers. This chapter will clarify what happens in situations where different processes are trying to write in the same row at the same time. It will also shed light on the behavior of read queries inside a transaction with the different types of isolation levels available in MySQL.

First, let's define the key concepts. A *transaction* is an operation performed (using one or more SQL statements) on a database as a single logical unit of work. All the SQL statements' modifications in a transaction are either committed (applied to the database) or rolled back (undone from the database) as a unit, never only partially. A database transaction must be atomic, consistent, isolated, and durable (the famous acronym *ACID*).

Locks are mechanisms used to ensure the integrity of the data stored in the database while applications and users are interacting with it. We will see that there are different types of lock, and some are more restrictive than others.

Databases would not need transactions and locks if requests were issued serially and processed in order, one at a time (a `SELECT` , then an `INSERT` , then an `UPDATE` , and so on). We illustrate this behavior in Figure 6-1.

However, the reality (fortunately!) is that MySQL can handle thousands of requests per second and process them in parallel, rather than serially. This chapter discusses what MySQL does to achieve this parallelism, for example, when requests to `SELECT` and `UPDATE` in the same row arrive simultaneously, or one arrives while the other is still executing. Figure 6-2 shows what this looks like.

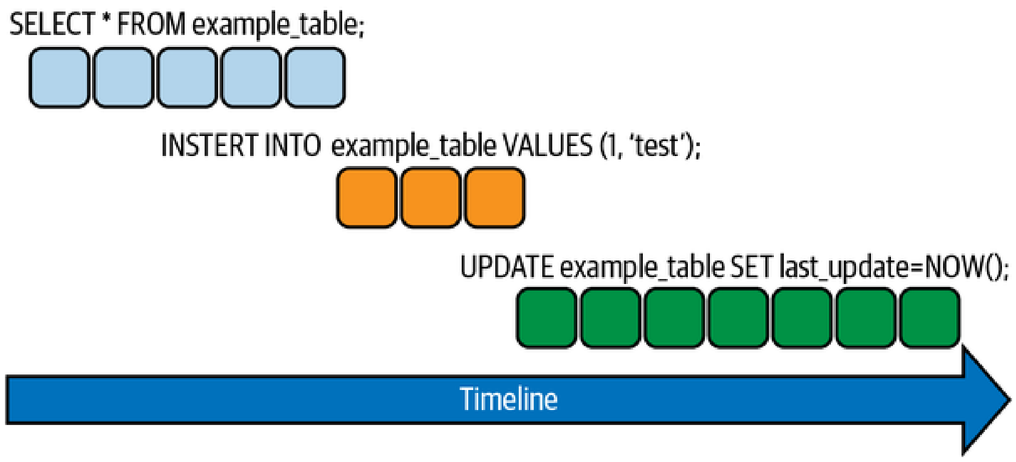
SELECT * FROM example_table;
INSTERT INTO example_table VALUES (1, 'test');
UPDATE example_table SET last_update=NOW();
Timeline

Figure 6-1. Serialized execution of SQL statements

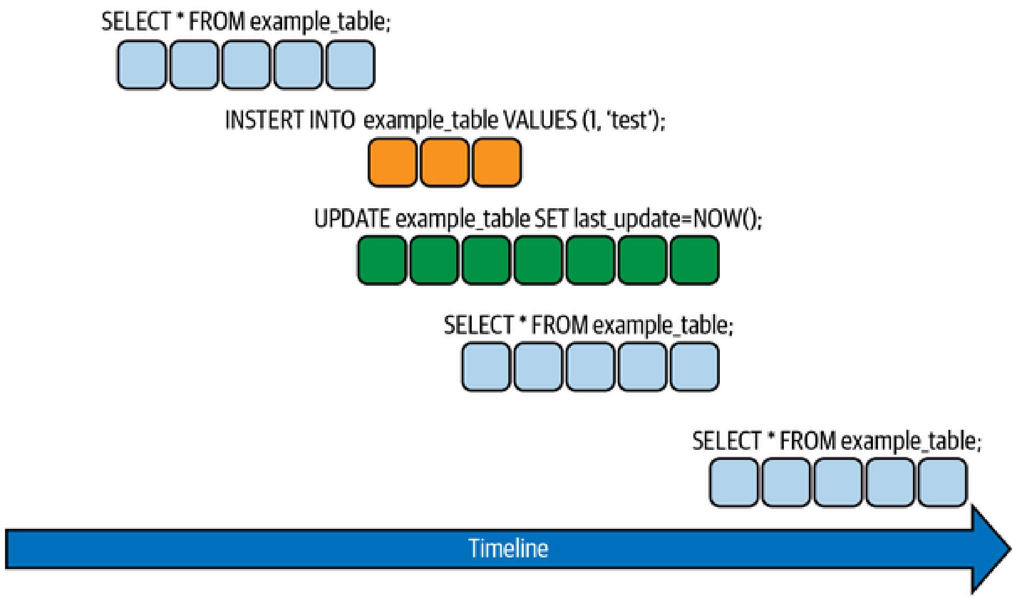
SELECT * FROM example_table;
INSTERT INTO example_table VALUES (1, 'test');
UPDATE example_table SET last_update=NOW();
SELECT * FROM example_table;
SELECT * FROM example_table;
Timeline

Figure 6-2. Parallel execution of SQL statements

For this chapter, we are particularly interested in how MySQL *isolates* the transactions (the *I* of ACID). We will show you common situations where locking occurs, investigate them, and discuss the MySQL parameters that control how much time a transaction can wait for a lock to be granted.

Isolation Levels

The *isolation level* is the setting that balances performance, reliability, consistency, and reproducibility of results when multiple transactions are making changes and performing queries simultaneously.

The SQL:1992 standard defines four classic isolation levels, and MySQL supports all of them. InnoDB supports each of the transaction isolation levels described here using different locking strategies. A user can also change the isolation level for a single session or all subsequent connections with the statement `SET [GLOBAL/SESSION] TRANSACTION`.

We can enforce a high degree of consistency with the default `REPEATABLE READ` isolation level for operations on data where ACID compliance is essential, and we can relax the consistency rules with `READ COMMITTED` or even `READ UNCOMMITTED` isolation in situations such as bulk reporting where precise consistency and repeatable results are less important than minimizing the amount of overhead for locking. `SERIALIZABLE` isolation enforces even stricter rules than `REPEATABLE READ` and is used mainly for special situations such as troubleshooting. Before diving into the details, let's take a look at some more terminology:

Dirty reads

These occur when a transaction is able to read data from a row that has been modified by another transaction that has not executed a `COMMIT` yet. If the transaction that made the modifications gets rolled back, the other one will have seen incorrect results that do not reflect the state of the database. Data integrity is compromised.

Non-repeatable reads

These occur when two queries in a transaction execute a `SELECT` and the values returned differ between the readings because of changes made by another transaction in the interim (if you read a row at time T1 and then try to read it again at time T2, the row may have been updated). The difference from a dirty read is that in this case there is a `COMMIT`. The initial `SELECT` query is not repeatable because it returns different values when issued the second time.

Phantom reads

These occur when a transaction is running, and another transaction adds rows to or deletes them from the records being read (again, in this case there is a `COMMIT` by the transaction modifying the data). This means that if the same query is executed again in the same transaction, it will return a different number of rows. Phantom reads can occur when there are no range locks guaranteeing the consistency of the data.

With those concepts in mind, let's take a closer look at the different isolation levels in MySQL.

REPEATABLE READ

`REPEATABLE READ` is the default isolation level for InnoDB. It ensures consistent reads within the same transaction—that is, that all queries within the transaction will see the same snapshot of the data, established by the first read. In this mode, InnoDB locks the index range scanned, using gap locks or next-key locks (described in "Locking") to block insertions by other sessions into any gaps within that range.

For example, suppose that in one session (session 1), we execute the following `SELECT`:

```
session1 > SELECT * FROM person WHERE i BETWEEN 1 AND 4
```

```
+---+----------+
| i | name     |
+---+----------+
| 1 | Vinicius |
| 2 | Sergey   |
| 3 | Iwo      |
| 4 | Peter    |
+---+----------+
4 rows in set (0.00 sec)
```

And in another session (session 2), we update the name in the second row:

```
session2 > UPDATE person SET name = 'Kuzmichev' WHERE i
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
session2> COMMIT;
```

```
Query OK, 0 rows affected (0.00 sec)
```

We can confirm the change in session 2:

```
session2 > SELECT * FROM person WHERE i BETWEEN 1 AND 4
```

```
+---+------------+
| i | name       |
+---+------------+
| 1 | Vinicius   |
| 2 | Kuzmichev  |
| 3 | Iwo        |
| 4 | Peter      |
+---+------------+
4 rows in set (0.00 sec)
```

But session 1 still shows the old value from its original snapshot of the data:

```
session1> SELECT * FROM person WHERE i BETWEEN 1 AND 4;
```

```
+---+----------+
| i | name     |
+---+----------+
| 1 | Vinicius |
| 2 | Sergey   |
| 3 | Iwo      |
| 4 | Peter    |
+---+----------+
```

With the `REPEATABLE READ` isolation level, there are thus no dirty reads and or non-repeatable reads. Each transaction reads the snapshot established by the first read.

READ COMMITTED

As a curiosity, the `READ COMMITTED` isolation level is the default for many databases, like Postgres, Oracle, and SQL Server, but not MySQL. So, those who are migrating to MySQL must be aware of this difference in the default behavior.

The main difference between `READ COMMITTED` and `REPEATABLE READ` is that with `READ COMMITTED` each consistent read, even within the same transaction, creates and reads its own fresh snapshot. This behavior can lead to *phantom reads* when executing multiple queries inside a transaction. Let's take a look at an example. In session 1, row 1 looks like this:

```
session1 > SELECT * FROM person WHERE i = 1;
```

```
+---+----------+
| i | name     |
+---+----------+
| 1 | Vinicius |
+---+----------+
1 row in set (0.00 sec)
```

Now suppose that in session 2 we update the first row of the `person` table and commit the transaction:

```
session2 > UPDATE person SET name = 'Grippa' WHERE i =
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
session2 > COMMIT;
```

```
Query OK, 0 rows affected (0.00 sec)
```

If we check session 1 again, we'll see that the value of the first row has changed:

```
session1 > SELECT * FROM person WHERE i = 1;
```

```
+---+--------+
| i | name   |
+---+--------+
| 1 | Grippa |
+---+--------+
```

The significant advantage of `READ COMMITTED` is that there are no gap locks, allowing the free insertion of new records next to locked records.

READ UNCOMMITTED

With the `READ UNCOMMITTED` isolation level MySQL performs `SELECT` statements in a non-locking fashion, which means two `SELECT` statements within the same transaction might not read the same version of a row. As we saw earlier, this phenomenon is called a dirty read. Consider how the previous example would play out using `READ UNCOMMITTED`. The main difference is that session 1 can see the results of session 2’s update *before* the commit. Let’s walk through another example. Suppose that in session 1 we execute the following `SELECT` statement:

```
session1 > SELECT * FROM person WHERE i = 5;
```

```
+---+---------+
| i | name    |
+---+---------+
| 5 | Marcelo |
+---+---------+
1 row in set (0.00 sec)
```

And in session 2, we perform this update *without* committing:

```
session2 > UPDATE person SET name = 'Altmann' WHERE i =
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

If we now perform the `SELECT` again in session 1, here’s what we’ll see:

```
session1 > SELECT * FROM person WHERE i = 5;
```

```
+---+---------+
| i | name    |
+---+---------+
| 5 | Altmann |
+---+---------+
1 row in set (0.00 sec)
```

We can see that session 1 can read the modified data even though it is in a transient state, and this change may end up being rolled back and not committed.

SERIALIZABLE

The most restricted isolation level available in MySQL is `SERIALIZABLE`. This is similar to `REPEATABLE READ`, but has an additional restriction of not allowing one transaction to interfere with another. So, with this locking mechanism, the inconsistent data scenario is no longer possible.

NOTE

For applications using `SERIALIZABLE`, it is important to have a retry strategy.

To make this clearer, imagine a finance database where we register customers' account balances in an `accounts` table. What will happen if two transactions try to update a customer's account balance at the same time? The following example illustrates this scenario. Assume that we have started two sessions using the default isolation level, `REPEATABLE READ`, and explicitly opened a transaction in each with `BEGIN`. In session 1, we select all the accounts in the `accounts` table:

```
session1> SELECT * FROM accounts;
```

```
+----+--------+---------+----------+-------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+-------------------
|  1 | Vinnie |      80 | USD      | 2021-07-13 20:39:2
|  2 | Sergey |     100 | USD      | 2021-07-13 20:39:3
|  3 | Markus |     100 | USD      | 2021-07-13 20:39:3
+----+--------+---------+----------+-------------------
3 rows in set (0.00 sec)
```

Then, in session 2, we select all accounts with balance of at least 80 USD:

```
session2> SELECT * FROM accounts WHERE balance >= 80;
```

```
+----+--------+---------+----------+------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+------------------
|  1 | Vinnie |      80 | USD      | 2021-07-13 20:39:2
|  2 | Sergey |     100 | USD      | 2021-07-13 20:39:3
|  3 | Markus |     100 | USD      | 2021-07-13 20:39:3
+----+--------+---------+----------+------------------
3 rows in set (0.00 sec)
```

Now, in session 1, we subtract 10 USD from account 1 and check the result:

```
session1> UPDATE accounts SET balance = balance - 10 WH
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

session1> SELECT * FROM accounts;
```

```
+----+--------+---------+----------+------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+------------------
|  1 | Vinnie |      70 | USD      | 2021-07-13 20:39:2
|  2 | Sergey |     100 | USD      | 2021-07-13 20:39:3
|  3 | Markus |     100 | USD      | 2021-07-13 20:39:3
+----+--------+---------+----------+------------------
3 rows in set (0.00 sec)
```

We can see that the balance of account 1 has decreased to 70 USD. So, we commit session 1 and then move to session 2 to see if it can read the new changes made by session 1:

```
session1> COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
session2> SELECT * FROM accounts WHERE id = 1;
```

```
+----+--------+---------+----------+-------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+-------------------
|  1 | Vinnie |      80 | USD      | 2021-07-13 20:39:2
+----+--------+---------+----------+-------------------
1 row in set (0.01 sec)
```

This `SELECT` query still returns the old data for account 1, with a balance of 80 USD, even though transaction 1 changed it to 70 USD and was committed successfully. That's because the `REPEATABLE READ` isolation level ensures that all read queries in a transaction are repeatable, which means they always return the same result, even if changes have been made by other committed transactions.

But what will happen if we also run the `UPDATE` query to subtract 10 USD from account 1's balance in session 2? Will it change the balance to 70 USD, or 60 USD, or throw an error? Let's see:

```
session2> UPDATE accounts SET balance = balance - 10 WH
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
session2> SELECT * FROM accounts WHERE id = 1;
```

```
+----+--------+---------+----------+-------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+-------------------
|  1 | Vinnie |      60 | USD      | 2021-07-13 20:39:2
+----+--------+---------+----------+-------------------
1 row in set (0.01 sec)
```

There's no error, and the account balance is now 60 USD, which is the correct value because transaction 1 has already committed the change that modified the balance to 70 USD.

However, from transaction 2's point of view, this doesn't make sense: in the last `SELECT` query it saw a balance of 80 USD, but after subtracting 10 USD from the account, now it sees a balance of 60 USD. The math doesn't work here because this transaction is still being affected by concurrent updates from other transactions.

This is the scenario where using `SERIALIZABLE` can help. Let's rewind to before we made any changes. This time we'll explicitly set the isolation level of both sessions to `SERIALIZABLE` with `SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE` before starting the transactions with `BEGIN`. Again, in session 1 we select all the accounts:

```
session1> SELECT * FROM accounts;
```

```
+----+--------+---------+----------+------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+------------------
|  1 | Vinnie |      80 | USD      | 2021-07-13 20:39:2
|  2 | Sergey |     100 | USD      | 2021-07-13 20:39:3
|  3 | Markus |     100 | USD      | 2021-07-13 20:39:3
+----+--------+---------+----------+------------------
3 rows in set (0.00 sec)
```

And in session 2 we select all the accounts with a balance greater than 80 USD:

```
session2> SELECT * FROM accounts WHERE balance >= 80;
```

```
+----+--------+---------+----------+------------------
| id | owner  | balance | currency | created_at
+----+--------+---------+----------+------------------
|  1 | Vinnie |      80 | USD      | 2021-07-13 20:39:2
|  2 | Sergey |     100 | USD      | 2021-07-13 20:39:3
|  3 | Markus |     100 | USD      | 2021-07-13 20:39:3
+----+--------+---------+----------+------------------
3 rows in set (0.00 sec)
```

Now, in session 1 we subtract 10 USD from account 1:

```
session1> UPDATE accounts SET balance = balance - 10 WH
```

And…nothing happens. This time the `UPDATE` query is blocked—the `SELECT` query in session 1 has locked those rows and prevents the `UPDATE` in session 2 from succeeding. Because we explicitly started our transactions with `BEGIN` (which has the same effect as disabling autocommit), InnoDB implicitly converts all plain `SELECT` statements in each transaction to `SELECT ... FOR SHARE`. It does not know ahead of time if the transaction will perform only reads or will modify rows, so InnoDB needs to place a lock on it to avoid the issue we demonstrated in the previous example. In this example, if autocommit were enabled, the `SELECT` query in session 2 would not block the update that we are trying to perform in session 1: MySQL would recognize that the query is a plain `SELECT` and does not need to block other queries because it is not going to modify any rows.

However, the update in session 2 will not hang forever; this lock has a timeout duration that is controlled by the `innodb_lock_wait_timeout` parameter. So, if session 1 doesn't commit or roll back its transaction to release the lock, once the session timeout is reached, MySQL will throw the following error:

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try res
```

Locking

Now that we've seen how each isolation level works, let's look at the different locking strategies InnoDB employs to implement them.

Locks are used in databases to protect shared resources or objects. They can act at different levels, such as:

- Table locking
- Metadata locking
- Row locking
- Application-level locking

MySQL uses metadata locking to manage concurrent access to database objects and to ensure data consistency. When there is an active transaction (explicit or implicit) on the table, MySQL does not allow writing of metadata (DDL statements, for example, update the metadata of the table). It does this to maintain metadata consistency in a concurrent environment.

If there is an active transaction (running, uncommitted, or rolled back) when a session performs one of the operations mentioned in the following list, the session requesting the data write will be held in the `Waiting for table metadata lock` status. A metadata lock wait may occur in any of the following scenarios:

- When you create or delete an index
- When you modify the table structure
- When you perform table maintenance operations (`OPTIMIZE TABLE` `REPAIR TABLE` , etc.)
- When you delete a table
- When you try to obtain a table-level write lock on the table (`LOCK TABLE table_name WRITE`)

To enable simultaneous write access by multiple sessions, InnoDB supports row-level locking.

Application-level or user-level locks, such as those provided by `GET_LOCK()` , can be used to simulate database locks such as record locks.

This book focuses on metadata and the row locks since they are the ones that affect the majority of users and are the most common.

Metadata Locks

The MySQL documentation provides the best definition of metadata locks:

To ensure transaction serializability, the server must not permit one session to perform a data definition language (DDL) statement on a table that is used in an uncompleted explicitly or implicitly started transaction in another session. The server achieves this by acquiring metadata locks on tables used within a transaction and deferring the locks' release until the transaction ends. A metadata lock on a table prevents changes to the table's structure. This locking approach has the implication that a table that is being used by a transaction within one session cannot be used in DDL statements by other sessions until the transaction ends.

With this definition in mind, let's take a look at metadata locking in action. First, we will create a dummy table and load some rows into it:

```
USE test;

DROP TABLE IF EXISTS `joinit`;

CREATE TABLE `joinit` (
  `i` int(11) NOT NULL AUTO_INCREMENT,
  `s` varchar(64) DEFAULT NULL,
  `t` time NOT NULL,
  `g` int(11) NOT NULL,
  PRIMARY KEY (`i`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1;

INSERT INTO joinit VALUES (NULL, uuid(), time(now()),
RAND( ) *60 )));
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
```

```
INSERT INTO joinit SELECT NULL, uuid(), time(now()),  (
FROM joinit;
```

Now that we have some dummy data, we will open one session (session 1) and execute an `UPDATE` :

```
session1> UPDATE joinit SET t=now();
```

Then, in a second session, we will try to add a new column to this table while the `UPDATE` is still running:

```
session2> ALTER TABLE joinit ADD COLUMN b INT;
```

And in a third session, we can execute the `SHOW PROCESSLIST` command to visualize the metadata lock:

```
session3> SHOW PROCESSLIST;


+----+----------+-----------+------+---------+------+..
| Id | User     | Host      | db   | Command | Time |..
+----+----------+-----------+------+---------+------+..
| 10 | msandbox | localhost | test | Query   |    3 |..
| 11 | msandbox | localhost | test | Query   |    1 |..
| 12 | msandbox | localhost | NULL | Query   |    0 |..
+----+----------+-----------+------+---------+------+..
...+----------------------------------+----------------
...| State                            | Info
...+----------------------------------+----------------
...| updating                         | UPDATE joinit SE
...| Waiting for table metadata lock  | ALTER TABLE joir
...| starting                         | SHOW PROCESSLIST
...+----------------------------------+----------------
...+-----------+---------------+
...| Rows_sent | Rows_examined |
...+-----------+---------------+
...|         0 |        179987 |
...|         0 |             0 |
...|         0 |             0 |
...+-----------+---------------+
```

Note that a long-running query or a query that is not using autocommit will have the same effect. For example, suppose we have an `UPDATE` running in session 1:

```
mysql > SET SESSION autocommit=0;

Query OK, 0 rows affected (0.00 sec)

mysql > UPDATE joinit SET t=NOW() LIMIT 1;

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

And we execute a DML statement in session 2:

```
mysql > ALTER TABLE joinit ADD COLUMN b INT;
```

If we check the process list in session 3, we can see the DDL waiting on the metadata lock (thread 11), while thread 10 has been sleeping since it executed the `UPDATE` (still not committed):

```
mysql > SHOW PROCESSLIST;
```

NOTE

MySQL is multithreaded, so there may be many clients issuing queries for a given table simultaneously. To minimize the problem with multiple client sessions having different states for the same table, each concurrent session opens the table independently. This uses additional memory but typically increases performance.

Before we start using the `sys` schema, it is necessary to enable MySQL instrumentation to monitor these locks. To do this, run the following command:

```
mysql> UPDATE performance_schema.setup_instruments SET
    -> WHERE NAME = 'wait/lock/metadata/sql/mdl';
```

```
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1 Changed: 0 Warnings: 0
```

The following query uses the `schema_table_lock_waits` view from the `sys` schema to illustrate how to observe metadata locks in the MySQL database:

```
mysql> SELECT *  FROM sys.schema_table_lock_waits;
```

This view displays which sessions are blocked waiting on metadata locks and what is blocking them. Rather than selecting all fields, the following example shows a more compact view:

```
mysql> SELECT object_name, waiting_thread_id, waiting_l
    -> waiting_query, sql_kill_blocking_query, blocking
    -> FROM sys.schema_table_lock_waits;
```

```
+-------------+-------------------+-------------------+
| object_name | waiting_thread_id | waiting_lock_type |
+-------------+-------------------+-------------------+
| joinit      |                29 | EXCLUSIVE         |
| joinit      |                29 | EXCLUSIVE         |
+-------------+-------------------+-------------------+
...+------------------------------------------------
...| waiting_query
...+------------------------------------------------
...| ALTER TABLE joinit ADD COLUMN  ...  CHAR(32) DEFAU
...| ALTER TABLE joinit ADD COLUMN  ...  CHAR(32) DEFAU
...|------------------------------------------------
```

```
...+-------------------------+-------------------+
...| sql_kill_blocking_query | blocking_thread_id |
...+-------------------------+-------------------+
...| KILL QUERY 3            |                29 |
...| KILL QUERY 5            |                31 |
...+-------------------------+-------------------+
2 rows in set (0.00 sec)
```

NOTE

The MySQL `sys` schema is a set of objects that helps DBAs and developers interpret data collected by the Performance Schema, a feature for monitoring MySQL Server execution at a low level. It is available for MySQL 5.7 and MySQL 8.0. If you want to use the `sys` schema in MySQL 5.6, it is possible to install it using the `sys` project available on GitHub:

```
# git clone https://github.com/mysql/mysql-sys.git
# cd mysql-sys/
# mysql -u root -p < ./sys_56.sql
```

Let's see what happens when we query the `metadata_locks` table:

```
mysql> SELECT * FROM performance_schema.metadata_locks\
```

```
*************************** 1. row *******************
          OBJECT_TYPE: GLOBAL
        OBJECT_SCHEMA: NULL
          OBJECT_NAME: NULL
OBJECT_INSTANCE_BEGIN: 140089691017472
            LOCK_TYPE: INTENTION_EXCLUSIVE
        LOCK_DURATION: STATEMENT
          LOCK_STATUS: GRANTED
               SOURCE:
      OWNER_THREAD_ID: 97
       OWNER_EVENT_ID: 34
...
*************************** 6. row *******************
          OBJECT_TYPE: TABLE
        OBJECT_SCHEMA: performance_schema
          OBJECT_NAME: metadata_locks
OBJECT_INSTANCE_BEGIN: 140089640911984
            LOCK_TYPE: SHARED_READ
        LOCK_DURATION: TRANSACTION
          LOCK_STATUS: GRANTED
               SOURCE:
      OWNER_THREAD_ID: 98
       OWNER_EVENT_ID: 10
6 rows in set (0.00 sec)
```

Note that a `SHARED_UPGRADABLE` lock is set on the `joinit` table, and an `EXCLUSIVE` lock is pending on the same table.

We can get a nice view of all metadata locks from other sessions, excluding our current one, with the following query:

```
mysql> SELECT object_type, object_schema, object_name,
    -> lock_status, thread_id, processlist_id, processl
    -> performance_schema.metadata_locks INNER JOIN per
    -> ON thread_id = owner_thread_id WHERE processlist
```

```
+-------------+---------------+-------------+---------
| OBJECT_TYPE | OBJECT_SCHEMA | OBJECT_NAME | LOCK_TYPE
+-------------+---------------+-------------+---------
| GLOBAL      | NULL          | NULL        | INTENTION
| SCHEMA      | test          | NULL        | INTENTION
| TABLE       | test          | joinit      | SHARED_UF
| BACKUP      | NULL          | NULL        | INTENTION
| TABLE       | test          | joinit      | EXCLUSIVE
+-------------+---------------+-------------+---------
...+-------------+-----------+----------------+...
...| LOCK_STATUS | THREAD_ID | PROCESSLIST_ID |...
...+-------------+-----------+----------------+...
...| GRANTED     |        97 |             71 |...
...| GRANTED     |        97 |             71 |...
...| GRANTED     |        97 |             71 |...
...| GRANTED     |        97 |             71 |...
...| PENDING     |        97 |             71 |...
...+-------------+-----------+----------------+...
...+-------------------------------------+
...| PROCESSLIST_INFO                    |
...+-------------------------------------+
...| alter table joinit add column b int |
...| alter table joinit add column b int |
...| alter table joinit add column b int |
...| alter table joinit add column b int |
...| alter table joinit add column b int |
...+-------------------------------------+
5 rows in set (0.00 sec)
```

If we look carefully, a DDL statement waiting for a query on its own is not a problem: it will have to wait until it can acquire the metadata lock, which is expected. The problem is that while waiting, it blocks every other query from accessing the resource.

We recommend the following actions to avoid long metadata locks:

- Perform DDL operations in non-busy times. This way you reduce the concurrency in the database between the regular application workload and the extra workload that the operation carries.
- Always use autocommit. MySQL has autocommit enabled by default. This will avoid transactions with pending commits.
- When performing a DDL operation, set a low value for `lock_wait_timeout` at the session level. Then, if the metadata lock can't be acquired, it won't block for a long time waiting. For example:

```
mysql> SET lock_wait_timeout = 3;
mysql> CREATE INDEX idx_1 ON example (col1);
```

You might also want to consider using the `pt-kill` tool to kill queries that have been running for a long time. For example, to kill queries that have been running for more than 60 seconds, issue this command:

```
$ pt-kill --busy-time 60 --kill
```

Row Locks

InnoDB implements standard row-level locking. This means that, in general terms, there are two types of locks:

- A *shared* (S) lock permits the transaction that holds the lock to read a row.
- An *exclusive* (X) lock permits the transaction that holds the lock to update or delete a row.

The names are self-explanatory: exclusive locks don't allow multiple transactions to acquire an exclusive lock in the same row while sharing a shared lock. That is why it is possible to have parallel reads for the same row, while parallel writes are not allowed.

InnoDB also supports multiple granularity locking, which permits the coexistence of row locks and table locks. Granular locking is possible due to the existence of *intention locks*, which are table-level locks that indicate which type of lock (shared or exclusive) a transaction requires later for a row in a table. There are two types of intention locks:

- An *intention shared* (IS) lock indicates that a transaction intends to set a shared lock on individual rows in a table.
- An *intention exclusive* (IX) lock indicates that a transaction intends to set an exclusive lock on individual rows in a table.

Before a transaction can acquire a shared or an exclusive lock, it is necessary to obtain the respective intention lock (IS or IX).

To make things a bit easier to understand, take a look at Table 6-1.

Table 6-1. Lock type compatibility matrix

	X	**IX**	**S**	**IS**
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

Another important concept is the *gap lock*, which is a lock on the gap between index records. Gap locks ensure that no new rows are added in the interval specified by the query; this means that when you run the same query twice, you get the same number of rows, regardless of other sessions' modifications to that table. They make the reads consistent and therefore make the replication between servers consistent. If you execute `SELECT * FROM example_table WHERE id > 1000 FOR UPDATE` twice, you expect to get the same result twice. To accomplish that, InnoDB locks all index records found by the `WHERE` clause with an exclusive lock and the gaps between them with a shared gap lock.

Let's see an example of a gap lock in action. First, we will execute a `SELECT` statement on the `person` table:

```
mysql> SELECT * FROM PERSON;
```

```
+----+-----------+
| i  | name      |
+----+-----------+
|  1 | Vinicius  |
|  2 | Kuzmichev |
|  3 | Iwo       |
|  4 | Peter     |
|  5 | Marcelo   |
|  6 | Guli      |
|  7 | Nando     |
| 10 | Jobin     |
| 15 | Rafa      |
| 18 | Leo       |
+----+-----------+
10 rows in set (0.00 sec)
```

Now, in session 1, we will perform a delete operation, but we will *not* commit:

```
session1> DELETE FROM person WHERE name LIKE 'Jobin';
```

```
Query OK, 1 row affected (0.00 sec)
```

And if we check in session 2, we can still see the row with Jobin:

```
session2> SELECT * FROM person;
```

```
+----+-----------+
| i  | name      |
+----+-----------+
|  1 | Vinicius  |
|  2 | Kuzmichev |
|  3 | Iwo       |
|  4 | Peter     |
|  5 | Marcelo   |
|  6 | Guli      |
|  7 | Nando     |
| 10 | Jobin     |
| 15 | Rafa       |
| 18 | Leo        |
+----+------------+
10 rows in set (0.00 sec)
```

The results show that there are gaps in the values of the primary key column that in theory are available to be used to insert new records. So what happens if we try to insert a new row with a value of 11? The insert will be locked and will fail:

```
transaction2 > INSERT INTO person VALUES (11, 'Bennie')
```

```
ERROR 1205 (HY000): Lockwait timeout exceeded; try rest
```

If we run `SHOW ENGINE INNODB STATUS`, we will see the locked transaction in the `TRANSACTIONS` section:

```
------- TRX HAS BEEN WAITING 17 SEC FOR THIS LOCK TO BE
RECORD LOCKS space id 28 page no 3 n bits 80 index PRIM
`test`.`person` trx id 4773 lock_mode X locks gap befor
intention waiting
```

Note that MySQL does not need gap locking for statements that lock rows using a unique index to search for a unique row. (This does not include the case where the search condition includes only some columns of a multiple-column unique index; in that case, gap locking does occur.) For example, if the `name` column has a unique index, the following `DELETE` statement uses only an index-record lock:

```
mysql> CREATE UNIQUE INDEX idx ON PERSON (name);
```

```
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> DELETE FROM person WHERE name LIKE 'Jobin';
```

```
Query OK, 1 row affected (0.00 sec)
```

Deadlocks

A *deadlock* is a situation where two (or more) competing actions are waiting for the other to finish. As a consequence, neither ever does. In computer science, the term refers to a specific condition where two or more processes are each waiting for another to release a resource. In this section, we will talk specifically about transaction deadlocks and how InnoDB solves this issue.

For a deadlock to happen, four conditions (known as the *Coffman conditions*) must exist:

1. *Mutual exclusion*. The process must hold at least one resource in a non-shareable mode. Otherwise, MySQL would not prevent the process from using the resource when necessary. Only one process can use the resource at any given moment in time.
2. *Hold and wait or resource holding*. A process is currently holding at least one resource and requesting additional resources held by other processes.
3. *No preemption*. A resource can be released only voluntarily by the process holding it.
4. *Circular wait*. Each process must be waiting for a resource held by another process, which in turn is waiting for the first process to release the resource.

Before moving on to an example, there are some misconceptions that you might hear and that it is essential to clarify. They are:

Transaction isolation levels are responsible for deadlocks.

> The possibility of deadlocks is not affected by the isolation level. The `READ COMMITTED` isolation level sets fewer locks, and hence it can help you avoid certain lock types (e.g., gap locking), but it won't prevent deadlocks entirely.

Small transactions are not affected by deadlocks.

> Small transactions are less prone to deadlocks because they run fast, so the chance of a conflict occurring is smaller than with more prolonged

operations. However, it can still happen if transactions do not use the same order of operations.

Deadlocks are terrible things.

It's problematic to have deadlocks in a database, but InnoDB can resolve them automatically, unless deadlock detection is disabled (by changing the value of `innodb_deadlock_detect`). A deadlock is a a bad situation, but resolution through the termination of one of the transactions ensures that processes cannot hold onto the resources for a long time, slowing or stalling the database completely until the offending query gets canceled by the `innodb_lock_wait_timeout` setting.

To illustrate deadlocks, we'll use the `world` database. If you need to import it, you can do so now by following the instructions in "Entity Relationship Modeling Examples".

Let's start by getting a list of Italian cities in the province of Toscana:

```
mysql> SELECT * FROM city WHERE CountryCode = 'ITA' ANI
```

```
+------+---------+-------------+----------+------------
| ID   | Name    | CountryCode | District | Population
+------+---------+-------------+----------+------------
| 1471 | Firenze | ITA         | Toscana  |     376662
| 1483 | Prato   | ITA         | Toscana  |     172473
| 1486 | Livorno | ITA         | Toscana  |     161673
| 1516 | Pisa    | ITA         | Toscana  |      92379
| 1518 | Arezzo  | ITA         | Toscana  |      91729
+------+---------+-------------+----------+------------
5 rows in set (0.00 sec)
```

Now let's say we have two transactions trying to update the populations of the same two cities in Toscana at the same time, but in different orders:

```
session1> UPDATE city SET Population=Population + 1 WHE
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
session2> UPDATE city SET Population=Population + 1 WHE
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
session1> UPDATE city SET Population=Population + 1 WHE
```

```
ERROR 1213 (40001): Deadlock found when trying to get l
```

```
session2> UPDATE city SET Population=Population + 1 WHE
```

```
Query OK, 1 row affected (5.15 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

And we had a deadlock in session 1. It is important to note that it is not always the second transaction that will fail. In this example, session 1 was the one that MySQL aborted. We can get information on the latest deadlock that happened in the MySQL database by running `SHOW ENGINE INNODB STATUS` :

```
mysql> SHOW ENGINE INNODB STATUS\G
```

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
2020-12-05 16:08:19 0x7f6949359700
*** (1) TRANSACTION:
TRANSACTION 10502342, ACTIVE 34 sec starting index reac
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1136, 2 row lock(
entries 1
```

```
MySQL thread id 71, OS thread handle 140090386671360, q
localhost msandbox updating
update city set Population=Population + 1 where ID = 14
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 6041 page no 15 n bits 248 index
`world`.`city` trx id 10502342 lock_mode X locks rec bu
*** (2) TRANSACTION:
TRANSACTION 10502341, ACTIVE 62 sec starting index read
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1136, 2 row lock(s), undo l
MySQL thread id 75, OS thread handle 140090176542464, q
localhost msandbox updating
update city set Population=Population + 1 where ID =151
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 6041 page no 15 n bits 248 index
`world`.`city` trx id 10502341 lock_mode X locks rec bu
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 6041 page no 16 n bits 248 index
`world`.`city` trx id 10502341 lock_mode X locks rec bu
*** WE ROLL BACK TRANSACTION (2)
...
```

If you want, you can log all the deadlocks that happen in MySQL in the MySQL error log. Using the `innodb_print_all_deadlocks` parameter, MySQL records all information about deadlocks from InnoDB user transactions in the error log. Otherwise, you see information about only the last deadlock using the `SHOW ENGINE INNODB STATUS` command.

MySQL Parameters Related to Isolation and Locks

To round out this chapter, let's take a look at a few MySQL parameters that are related to isolation behavior and lock duration:

`transaction_isolation`

Sets the transaction isolation level. This parameter can change the behavior at the `GLOBAL`, `SESSION`, or `NEXT_TRANSACTION` level:

```
mysql> SET SESSION transaction_isolation='READ-COMMITTE
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SHOW SESSION VARIABLES LIKE '%isol%';
```

```
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
| tx_isolation          | READ-COMMITTED |
+-----------------------+----------------+
```

NOTE

`transaction_isolation` was added in MySQL 5.7.20 as a synonym for `tx_isolation`, which is now deprecated and has been removed in MySQL 8.0. Applications should be adjusted to use `transaction_isolation` in preference to `tx_isolation`.

innodb_lock_wait_timeout

Specifies the amount of time in seconds an InnoDB transaction waits for a row lock before giving up. The default value is 50 seconds. The transaction raises the following error if the time waiting for the lock exceeds the `innodb_lock_wait_timeout` value:

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try res
```

innodb_print_all_deadlocks

Causes MySQL to record information about all deadlocks resulting from InnoDB user transactions in the MySQL error log. We can enable this dynamically with the following command:

```
mysql> SET GLOBAL innodb_print_all_deadlocks = 1;
```

lock_wait_timeout

Specifies the timeout in seconds for attempts to acquire metadata locks. To avoid long metadata locks stalling the database, we can set `lock_wait_timeout=1` at the session level before executing the DDL statement. In this case, if the operation can't acquire the lock, it will give up and let other requests execute. For example:

```
mysql> SET SESSION lock_wait_timeout=1;
mysql> CREATE TABLE t1(i INT NOT NULL AUTO_INCREMENT PF
    -> ENGINE=InnoDB;
```

innodb_deadlock_detect

Disables deadlock monitoring. Note that this only means that MySQL will not kill a query to undo the deadlock knot. Disabling deadlock detection will *not* prevent deadlocks from happening, but it will make MySQL rely on the `innodb_lock_wait_timeout` setting for transaction rollback when a deadlock occurs.