committed rows, even though the session could not query them. If a transaction does update or delete rows committed by a different transaction, those changes do become visible to the current transaction. For example, you might encounter a situation like the following:

SELECT COUNT(c1) FROM t1 WHERE c1 = 'xyz';

-- Returns 0: no rows match .

DELETE FROM t1 WHERE c1 = 'xyz';

-- Deletes several rows recently committed by other transaction.

SELECT COUNT(c2) FROM t1 WHERE c2 = 'abc';

-- Returns 0: no rows match .

UPDATE t1 SET c2 = 'cba' WHERE c2 = 'abc';

-- Affects 10 rows: another txn just committed 10 rows with 'abc' values .

SELECT COUNT(c2) FROM t1 WHERE c2 = 'cba';

-- Returns 10: this txn can now see the rows it just updated.

You can advance your timepoint by committing your transaction and then doing another SELECT or

START TRANSACTION WITH CONSISTENT SNAPSHOT.

This is called *multi-versioned* *concurrency* *control*.

In the following example, session A sees the row inserted by B only when B has committed the insert and A has committed as well, so that the timepoint is advanced past the commit of B.

|  |  |  |
| --- | --- | --- |
| time  |  |  |  |  v | Session A  SET autocommit=0;  SELECT \* FROM t;  empty set  SELECT \* FROM t;  empty set  SELECT \* FROM t;  empty set  COMMIT;  SELECT \* FROM t;  ---------------------  | 1 | 2 |  --------------------- | Session B  SET autocommit=0;  INSERT INTO t VALUES (1, 2);  COMMIT; |

If you want to see the “freshest” state of the database, use either the READ COMMITTED isolation level or a locking read:

SELECT \* FROM t FOR SHARE;

With READ COMMITTED isolation level, each consistent read within a transaction sets and reads its own fresh snapshot. With FOR SHARE, a locking read occurs instead: A SELECT blocks until the transaction containing the freshest rows ends (see [Section 15.7.2.4, “Locking Reads”](#_bookmark1)).

Consistent read does not work over certain DDL statements:

• Consistent read does not work over DROP TABLE, because MySQL cannot use a table that has been dropped and InnoDB destroys the table.

• Consistent read does not work over ALTER TABLE operations that make a temporary copy of the original table and delete the original table when the temporary copy is built. When you reissue a consistent read within a transaction, rows in the new table are not visible because those rows did not exist when the transaction's snapshot was taken. In this case, the transaction returns an error: [ER\_TABLE\_DEF\_CHANGED](https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html#error_er_table_def_changed), “Table definition has changed, please retry transaction” .



The type of read varies for selects in clauses like INSERT INTO ... SELECT, UPDATE ... (SELECT), and CREATE TABLE ... SELECT that do not specify FOR UPDATE or FOR SHARE:

• By default, InnoDB uses stronger locks for those statements and the SELECT part acts like READ COMMITTED, where each consistent read, even within the same transaction, sets and reads its own fresh snapshot.

• To perform a nonlocking read in such cases, set the isolation level of the transaction to READ UNCOMMITTED or READ COMMITTED to avoid setting locks on rows read from the selected table.

**15.7.2.4** **Locking** **Reads**

If you query data and then insert or update related data within the same transaction, the regular SELECT statement does not give enough protection. Other transactions can update or delete the same rows you just queried. InnoDB supports two types of locking reads that offer extra safety:

• SELECT ... FOR SHARE

Sets a shared mode lock on any rows that are read. Other sessions can read the rows, but

cannot modify them until your transaction commits. If any of these rows were changed by another transaction that has not yet committed, your query waits until that transaction ends and then uses the latest values.

**Note**

SELECT ... FOR SHARE is a replacement for SELECT ... LOCK IN SHARE MODE, but LOCK IN SHARE MODE remains available for backward compatibility. The statements are equivalent. However, FOR SHARE supports OF *table\_name*, NOWAIT, and SKIP LOCKED options. See [Locking Read](#_bookmark2) [Concurrency with NOWAIT and SKIP LOCKED](#_bookmark2).

Prior to MySQL 8.0.22, SELECT ... FOR SHARE requires the SELECT privilege and at least one of the DELETE, LOCK TABLES, or UPDATE privileges. From MySQL 8.0.22, only the SELECT privilege is required.

From MySQL 8.0.22, SELECT ... FOR SHARE statements do not acquire read locks on MySQL grant tables. For more information, see Grant Table Concurrency.

• SELECT ... FOR UPDATE

For index records the search encounters, locks the rows and any associated index entries, the same as if you issued an UPDATE statement for those rows. Other transactions are blocked from updating those rows, from doing SELECT ... FOR SHARE, or from reading the data in certain transaction isolation levels. Consistent reads ignore any locks set on the records that exist in the read view. (Old versions of a record cannot be locked; they are reconstructed by applying undo logs on an in- memory copy of the record.)

SELECT ... FOR UPDATE requires the SELECT privilege and at least one of the DELETE, LOCK TABLES, or UPDATE privileges.

These clauses are primarily useful when dealing with tree-structured or graph-structured data, either in a single table or split across multiple tables. You traverse edges or tree branches from one place to another, while reserving the right to come back and change any of these “pointer” values.

All locks set by FOR SHARE and FOR UPDATE queries are released when the transaction is committed or rolled back.

**Note**

Locking reads are only possible when autocommit is disabled (either by beginning transaction with START TRANSACTION or by setting autocommit to

0.

A locking read clause in an outer statement does not lock the rows of a table in a nested subquery unless a locking read clause is also specified in the subquery. For example, the following statement does not lock rows in table t2.

SELECT \* FROM t1 WHERE c1 = (SELECT c1 FROM t2) FOR UPDATE; To lock rows in table t2, add a locking read clause to the subquery: SELECT \* FROM t1 WHERE c1 = (SELECT c1 FROM t2 FOR UPDATE) FOR UPDATE;

**Locking** **Read** **Examples**

Suppose that you want to insert a new row into a table child, and make sure that the child row has a parent row in table parent. Your application code can ensure referential integrity throughout this sequence of operations.

First, use a consistent read to query the table PARENT and verify that the parent row exists. Can you safely insert the child row to table CHILD? No, because some other session could delete the parent row in the moment between your SELECT and your INSERT, without you being aware of it.

To avoid this potential issue, perform the SELECT using FOR SHARE: SELECT \* FROM parent WHERE NAME = 'Jones' FOR SHARE;

After the FOR SHARE query returns the parent 'Jones', you can safely add the child record to the CHILD table and commit the transaction. Any transaction that tries to acquire an exclusive lock in the applicable row in the PARENT table waits until you are finished, that is, until the data in all tables is in a consistent state.

For another example, consider an integer counter field in a table CHILD\_CODES, used to assign a unique identifier to each child added to table CHILD. Do not use either consistent read or a shared mode read to read the present value of the counter, because two users of the database could see the same value for the counter, and a duplicate-key error occurs if two transactions attempt to add rows with the same identifier to the CHILD table.

Here, FOR SHARE is not a good solution because if two users read the counter at the same time, at least one of them ends up in deadlock when it attempts to update the counter.

To implement reading and incrementing the counter, first perform a locking read of the counter using FOR UPDATE, and then increment the counter. For example:

SELECT counter\_field FROM child\_codes FOR UPDATE;

UPDATE child\_codes SET counter\_field = counter\_field + 1;

A SELECT ... FOR UPDATE reads the latest available data, setting exclusive locks on each row it reads. Thus, it sets the same locks a searched SQL UPDATE would set on the rows.

The preceding description is merely an example of how SELECT ... FOR UPDATE works. In MySQL, the specific task of generating a unique identifier actually can be accomplished using only a single access to the table:

UPDATE child\_codes SET counter\_field = LAST\_INSERT\_ID(counter\_field + 1);

SELECT LAST\_INSERT\_ID();

The SELECT statement merely retrieves the identifier information (specific to the current connection). It does not access any table.

**Locking** **Read** **Concurrency** **with** **NOWAIT** **and** **SKIP** **LOCKED**

If a row is locked by a transaction, a SELECT ... FOR UPDATE or SELECT ... FOR SHARE transaction that requests the same locked row must wait until the blocking transaction releases the row



lock. This behavior prevents transactions from updating or deleting rows that are queried for updates by other transactions. However, waiting for a row lock to be released is not necessary if you want the query to return immediately when a requested row is locked, or if excluding locked rows from the result set is acceptable.

To avoid waiting for other transactions to release row locks, NOWAIT and SKIP LOCKED options may be used with SELECT ... FOR UPDATE or SELECT ... FOR SHARE locking read statements.

• NOWAIT

A locking read that uses NOWAIT never waits to acquire a row lock. The query executes immediately, failing with an error if a requested row is locked.

• SKIP LOCKED

A locking read that uses SKIP LOCKED never waits to acquire a row lock. The query executes immediately, removing locked rows from the result set.

**Note**

Queries that skip locked rows return an inconsistent view of the data. SKIP LOCKED is therefore not suitable for general transactional work. However, it may be used to avoid lock contention when multiple sessions access the same queue-like table.

NOWAIT and SKIP LOCKED only apply to row-level locks.

Statements that use NOWAIT or SKIP LOCKED are unsafe for statement based replication.

The following example demonstrates NOWAIT and SKIP LOCKED. Session 1 starts a transaction that takes a row lock on a single record. Session 2 attempts a locking read on the same record using the NOWAIT option. Because the requested row is locked by Session 1, the locking read returns immediately with an error. In Session 3, the locking read with SKIP LOCKED returns the requested rows except for the row that is locked by Session 1.

# Session 1:

mysql> **CREATE** **TABLE** **t** **(i** **INT,** **PRIMARY** **KEY** **(i))** **ENGINE** **=** **InnoDB;**

mysql> **INSERT** **INTO** **t** **(i)** **VALUES(1),(2),(3);**

mysql> **START** **TRANSACTION;**

mysql> **SELECT** **\*** **FROM** **t** **WHERE** **i** **=** **2** **FOR** **UPDATE;**

+---+

| i |

+---+

| 2 |

+---+

# Session 2:

mysql> **START** **TRANSACTION;**

mysql> **SELECT** **\*** **FROM** **t** **WHERE** **i** **=** **2** **FOR** **UPDATE** **NOWAIT;**

ERROR 3572 (HY000): Do not wait for lock.

# Session 3:

mysql> **START** **TRANSACTION;**

mysql> **SELECT** **\*** **FROM** **t** **FOR** **UPDATE** **SKIP** **LOCKED;**

+---+

| i |

+---+

| 1 |

| 3 |

+---+

**15.7.3** **Locks** **Set** **by** **Different** **SQL** **Statements** **in** **InnoDB**

A locking read, an UPDATE, or a DELETE generally set record locks on every index record that is scanned in the processing of an SQL statement. It does not matter whether there are WHERE conditions in the statement that would exclude the row. InnoDB does not remember the exact WHERE condition, but only knows which index ranges were scanned. The locks are normally next- key locks that also block inserts into the “gap” immediately before the record. However, gap locking can be disabled explicitly, which causes next-key locking not to be used. For more information, see Section 15.7.1, “InnoDB Locking” . The transaction isolation level can also affect which locks are set; see Section 15.7.2.1, “Transaction Isolation Levels” .

If a secondary index is used in a search and the index record locks to be set are exclusive, InnoDB also retrieves the corresponding clustered index records and sets locks on them.

If you have no indexes suitable for your statement and MySQL must scan the entire table to process the statement, every row of the table becomes locked, which in turn blocks all inserts by other users to the table. It is important to create good indexes so that your queries do not scan more rows than necessary.

InnoDB sets specific types of locks as follows.

• 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. However, only an index record lock is required for statements that lock rows using a unique index to search for a unique row.

• SELECT ... FOR UPDATE and SELECT ... FOR SHARE statements that use a unique index acquire locks for scanned rows, and release the locks for rows that do not qualify for inclusion in the result set (for example, if they do not meet the criteria given in the WHERE clause). However, in some cases, rows might not be unlocked immediately because the relationship between a result row and its original source is lost during query execution. For example, in a UNION, scanned (and locked) rows from a table might be inserted into a temporary table before evaluating whether they qualify for the result set. In this circumstance, the relationship of the rows in the temporary table to the rows in the original table is lost and the latter rows are not unlocked until the end of query execution.

• For locking reads (SELECT with FOR UPDATE or FOR SHARE), UPDATE, and DELETE statements, the locks that are taken depend on whether the statement uses a unique index with a unique search condition or a range-type search condition.

• For a unique index with a unique search condition, InnoDB locks only the index record found, not the gap before it.

• For other search conditions, and for non-unique indexes, InnoDB locks the index range scanned, using gap locks or next-key locks to block insertions by other sessions into the gaps covered by the range. For information about gap locks and next-key locks, see Section 15.7.1, “InnoDB Locking” .

• For index records the search encounters, SELECT ... FOR UPDATE blocks other sessions from doing SELECT ... FOR SHARE or from reading in certain transaction isolation levels. Consistent reads ignore any locks set on the records that exist in the read view.

• UPDATE ... WHERE ... sets an exclusive next-key lock on every record the search encounters. However, only an index record lock is required for statements that lock rows using a unique index to search for a unique row.

• When UPDATE modifies a clustered index record, implicit locks are taken on affected secondary index records. The UPDATE operation also takes shared locks on affected secondary index records

when performing duplicate check scans prior to inserting new secondary index records, and when inserting new secondary index records.

• DELETE FROM ... WHERE ... sets an exclusive next-key lock on every record the search encounters. However, only an index record lock is required for statements that lock rows using a unique index to search for a unique row.

• INSERT sets an exclusive lock on the inserted row. This lock is an index-record lock, not a next-key lock (that is, there is no gap lock) and does not prevent other sessions from inserting into the gap before the inserted row.

Prior to inserting the row, a type of gap lock called an insert intention gap lock is set. This lock signals 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. Suppose that there are index records with values of 4 and 7. Separate transactions that attempt to insert values of 5 and 6 each lock the gap between 4 and 7 with insert intention locks prior to obtaining the exclusive lock on the inserted row, but do not block each other because the rows are nonconflicting.

If a duplicate-key error occurs, a shared lock on the duplicate index record is set. This use of a shared lock can result in deadlock should there be multiple sessions trying to insert the same row if another session already has an exclusive lock. This can occur if another session deletes the row. Suppose that an InnoDB table t1 has the following structure:

CREATE TABLE t1 (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;

Now suppose that three sessions perform the following operations in order:

Session 1:

START TRANSACTION;

INSERT INTO t1 VALUES(1);

Session 2:

START TRANSACTION;

INSERT INTO t1 VALUES(1);

Session 3:

START TRANSACTION;

INSERT INTO t1 VALUES(1);

Session 1:

ROLLBACK;

The first operation by session 1 acquires an exclusive lock for the row. The operations by sessions 2 and 3 both result in a duplicate-key error and they both request a shared lock for the row. When session 1 rolls back, it releases its exclusive lock on the row and the queued shared lock requests for sessions 2 and 3 are granted. At this point, sessions 2 and 3 deadlock: Neither can acquire an exclusive lock for the row because of the shared lock held by the other.

A similar situation occurs if the table already contains a row with key value 1 and three sessions perform the following operations in order:

Session 1:

START TRANSACTION;

DELETE FROM t1 WHERE i = 1;

Session 2:

START TRANSACTION;

INSERT INTO t1 VALUES(1);

Session 3:

START TRANSACTION;

INSERT INTO t1 VALUES(1);

Session 1:

COMMIT;

The first operation by session 1 acquires an exclusive lock for the row. The operations by sessions 2 and 3 both result in a duplicate-key error and they both request a shared lock for the row. When session 1 commits, it releases its exclusive lock on the row and the queued shared lock requests for sessions 2 and 3 are granted. At this point, sessions 2 and 3 deadlock: Neither can acquire an exclusive lock for the row because of the shared lock held by the other.

• INSERT ... ON DUPLICATE KEY UPDATE differs from a simple INSERT in that an exclusive lock rather than a shared lock is placed on the row to be updated when a duplicate-key error occurs. An exclusive index-record lock is taken for a duplicate primary key value. An exclusive next-key lock is taken for a duplicate unique key value.

• 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 to be replaced.

• INSERT INTO T SELECT ... FROM S WHERE ... sets an exclusive index record lock (without a gap lock) on each row inserted into T. If the transaction isolation level is READ COMMITTED, InnoDB does the search on S as a consistent read (no locks). Otherwise, InnoDB sets shared next- key locks on rows from S. InnoDB has to set locks in the latter case: During roll-forward recovery using a statement-based binary log, every SQL statement must be executed in exactly the same way it was done originally.

CREATE TABLE ... SELECT ... performs the SELECT with shared next-key locks or as a consistent read, as for INSERT ... SELECT.

When a SELECT is used in the constructs REPLACE INTO t SELECT ... FROM s WHERE ... or UPDATE t ... WHERE col IN (SELECT ... FROM s ...), InnoDB sets shared next-key locks on rows from table s.

• InnoDB sets an exclusive lock on the end of the index associated with the AUTO\_INCREMENT column while initializing a previously specified AUTO\_INCREMENT column on a table.

With innodb\_autoinc\_lock\_mode=0, InnoDB uses a special AUTO-INC table lock mode where the lock is obtained and held to the end of the current SQL statement (not to the end of the entire transaction) while accessing the auto-increment counter. Other clients cannot insert into the table while the AUTO-INC table lock is held. The same behavior occurs for “bulk inserts” with innodb\_autoinc\_lock\_mode=1. Table-level AUTO-INC locks are not used with innodb\_autoinc\_lock\_mode=2. For more information, See Section 15.6.1.6, “AUTO\_INCREMENT Handling in InnoDB” .

InnoDB fetches the value of a previously initialized AUTO\_INCREMENT column without setting any locks.

• 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.

Otherwise, InnoDB's automatic deadlock detection cannot detect deadlocks where such table locks are involved. Also, because in this case the higher MySQL layer does not know about row-level

locks, it is possible to get a table lock on a table where another session currently has row-level locks. However, this does not endanger transaction integrity, as discussed in [Section 15.7.5.2, “Deadlock](#_bookmark3) [Detection”](#_bookmark3) .

• LOCK TABLES acquires two locks on each table if innodb\_table\_locks=1 (the default). In addition to a table lock on the MySQL layer, it also acquires an InnoDB table lock. To avoid acquiring InnoDB table locks, set innodb\_table\_locks=0. If no InnoDB table lock is acquired, LOCK TABLES completes even if some records of the tables are being locked by other transactions.

In MySQL 8.0, innodb\_table\_locks=0 has no effect for tables locked explicitly with LOCK TABLES ... WRITE. It does have an effect for tables locked for read or write by LOCK TABLES ... WRITE implicitly (for example, through triggers) or by LOCK TABLES ... READ.

• All InnoDB locks held by a transaction are released when the transaction is committed or aborted. Thus, it does not make much sense to invoke LOCK TABLES on InnoDB tables in autocommit=1 mode because the acquired InnoDB table locks would be released immediately.

• You cannot lock additional tables in the middle of a transaction because LOCK TABLES performs an implicit COMMIT and UNLOCK TABLES.

**15.7.4** **Phantom** **Rows**

The so-called *phantom* problem occurs within a transaction when the same query produces different sets of rows at different times. For example, if a SELECT is executed twice, but returns a row the second time that was not returned the first time, the row is a “phantom” row.

Suppose that there is an index on the id column of the child table and that you want to read and lock all rows from the table having an identifier value larger than 100, with the intention of updating some column in the selected rows later:

SELECT \* FROM child WHERE id > 100 FOR UPDATE;

The query scans the index starting from the first record where id is bigger than 100. Let the table contain rows having id values of 90 and 102. If the locks set on the index records in the scanned range do not lock out inserts made in the gaps (in this case, the gap between 90 and 102), another session can insert a new row into the table with an id of 101. If you were to execute the same SELECT within the same transaction, you would see a new row with an id of 101 (a “phantom”) in the result set returned by the query. If we regard a set of rows as a data item, the new phantom child would violate the isolation principle of transactions that a transaction should be able to run so that the data it has read does not change during the transaction.

To prevent phantoms, InnoDB uses an algorithm called *next-key* *locking* that combines index-row locking with gap locking. InnoDB performs row-level locking in such a way that when it searches or scans a table index, it sets shared or exclusive locks on the index records it encounters. Thus, the row- level locks are actually index-record locks. In addition, a next-key lock on an index record also affects the “gap” before the index record. That is, a next-key lock is an index-record lock plus a gap lock on the gap preceding the index record. If one session has a shared or exclusive lock on record R in an index, another session cannot insert a new index record in the gap immediately before R in the index order.

When InnoDB scans an index, it can also lock the gap after the last record in the index. Just that happens in the preceding example: To prevent any insert into the table where id would be bigger than 100, the locks set by InnoDB include a lock on the gap following id value 102.

You can use next-key locking to implement a uniqueness check in your application: If you read your data in share mode and do not see a duplicate for a row you are going to insert, then you can safely insert your row and know that the next-key lock set on the successor of your row during the read prevents anyone meanwhile inserting a duplicate for your row. Thus, the next-key locking enables you to “lock” the nonexistence of something in your table.

Gap locking can be disabled as discussed in Section 15.7.1, “InnoDB Locking” . This may cause phantom problems because other sessions can insert new rows into the gaps when gap locking is disabled.

**15.7.5** **Deadlocks** **in** **InnoDB**

A deadlock is a situation where different transactions are unable to proceed because each holds a lock that the other needs. Because both transactions are waiting for a resource to become available, neither ever release the locks it holds.

A deadlock can occur when transactions lock rows in multiple tables (through statements such as UPDATE or SELECT ... FOR UPDATE), but in the opposite order. A deadlock can also occur when such statements lock ranges of index records and gaps, with each transaction acquiring some locks but not others due to a timing issue. For a deadlock example, see [Section 15.7.5.1, “An InnoDB Deadlock](#_bookmark5) [Example”](#_bookmark5) .

To reduce the possibility of deadlocks, use transactions rather than LOCK TABLES statements; keep transactions that insert or update data small enough that they do not stay open for long periods of time; when different transactions update multiple tables or large ranges of rows, use the same order of operations (such as SELECT ... FOR UPDATE) in each transaction; create indexes on the columns used in SELECT ... FOR UPDATE and UPDATE ... WHERE statements. The possibility of deadlocks is not affected by the isolation level, because the isolation level changes the behavior of read operations, while deadlocks occur because of write operations. For more information about avoiding and recovering from deadlock conditions, see [Section 15.7.5.3, “How to Minimize and Handle](#_bookmark6) [Deadlocks”](#_bookmark6) .

When deadlock detection is enabled (the default) and a deadlock does occur, InnoDB detects the condition and rolls back one of the transactions (the victim). If deadlock detection is disabled using the innodb\_deadlock\_detect variable, InnoDB relies on the innodb\_lock\_wait\_timeout setting to roll back transactions in case of a deadlock. Thus, even if your application logic is correct, you must still handle the case where a transaction must be retried. To view the last deadlock in an InnoDB user transaction, use SHOW ENGINE INNODB STATUS. If frequent deadlocks highlight a problem with transaction structure or application error handling, enable innodb\_print\_all\_deadlocks to print information about all deadlocks to the mysqld error log. For more information about how deadlocks are automatically detected and handled, see [Section 15.7.5.2, “Deadlock Detection”](#_bookmark3) .

**15.7.5.1** **An** **InnoDB** **Deadlock** **Example**

The following example illustrates how an error can occur when a lock request causes a deadlock. The example involves two clients, A and B.

InnoDB status contains details of the last deadlock. For frequent deadlocks, enable global variable innodb\_print\_all\_deadlocks. This adds deadlock information to the error log.

Client A enables innodb\_print\_all\_deadlocks, creates two tables, 'Animals' and 'Birds', and inserts data into each. Client A begins a transaction, and selects a row in Animals in share mode:

mysql> **SET** **GLOBAL** **innodb\_print\_all\_deadlocks** **=** **ON;**

Query OK, 0 rows affected (0.00 sec)

mysql> **CREATE** **TABLE** **Animals** **(name** **VARCHAR(10)** **PRIMARY** **KEY,** **value** **INT)** **ENGINE** **=** **InnoDB;**

Query OK, 0 rows affected (0.01 sec)

mysql> **CREATE** **TABLE** **Birds** **(name** **VARCHAR(10)** **PRIMARY** **KEY,** **value** **INT)** **ENGINE** **=** **InnoDB;**

Query OK, 0 rows affected (0.01 sec)

mysql> **INSERT** **INTO** **Animals** **(name,value)** **VALUES** **("Aardvark",10);**

Query OK, 1 row affected (0.00 sec)

mysql> **INSERT** **INTO** **Birds** **(name,value)** **VALUES** **("Buzzard",20);**

Query OK, 1 row affected (0.00 sec)

mysql> **START** **TRANSACTION;**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **value** **FROM** **Animals** **WHERE** **name='Aardvark'** **FOR** **SHARE;**

+-------+

| value |

+-------+

| 10 |

+-------+

1 row in set (0.00 sec)

Next, client B begins a transaction, and selects a row in Birds in share mode:

mysql> **START** **TRANSACTION;**

Query OK, 0 rows affected (0.00 sec)

mysql> **SELECT** **value** **FROM** **Birds** **WHERE** **name='Buzzard'** **FOR** **SHARE;**

+-------+

| value |

+-------+

| 20 |

+-------+

1 row in set (0.00 sec)

The Performance Schema shows the locks after the two select statements:

mysql> **SELECT** **ENGINE\_TRANSACTION\_ID** **as** **Trx\_Id,**

**OBJECT\_NAME** **as** **`Table`,**

**INDEX\_NAME** **as** **`Index`,**

**LOCK\_DATA** **as** **Data,**

**LOCK\_MODE** **as** **Mode,**

**LOCK\_STATUS** **as** **Status,**

**LOCK\_TYPE** **as** **Type**

**FROM** **performance\_schema** **.data\_locks;**

+-----------------+---------+---------+------------+---------------+---------+--------+

| Trx\_Id | Table | Index | Data | Mode | Status | Type |

+-----------------+---------+---------+------------+---------------+---------+--------+

| 421291106147544 | Animals | NULL | NULL | IS | GRANTED | TABLE |

| 421291106147544 | Animals | PRIMARY | 'Aardvark' | S,REC\_NOT\_GAP | GRANTED | RECORD |

| 421291106148352 | Birds | NULL | NULL | IS | GRANTED | TABLE |

| 421291106148352 | Birds | PRIMARY | 'Buzzard' | S,REC\_NOT\_GAP | GRANTED | RECORD |

+-----------------+---------+---------+------------+---------------+---------+--------+

4 rows in set (0.00 sec)

Client B then updates a row in Animals:

mysql> **UPDATE** **Animals** **SET** **value=30** **WHERE** **name='Aardvark';**

Client B has to wait. The Performance Schema shows the wait for a lock:

mysql> **SELECT** **REQUESTING\_ENGINE\_LOCK\_ID** **as** **Req\_Lock\_Id,**

**REQUESTING\_ENGINE\_TRANSACTION\_ID** **as** **Req\_Trx\_Id,**

**BLOCKING\_ENGINE\_LOCK\_ID** **as** **Blk\_Lock\_Id,**

**BLOCKING\_ENGINE\_TRANSACTION\_ID** **as** **Blk\_Trx\_Id**

**FROM** **performance\_schema** **.data\_lock\_waits;**

+ +------------+----------------------------------------+----------

| Req\_Lock\_Id | Req\_Trx\_Id | Blk\_Lock\_Id | Blk\_Trx\_I

+ +------------+----------------------------------------+----------

| 139816129437696:27:4:2:139816016601240 | 43260 | 139816129436888:27:4:2:139816016594720 | 421291106

+ +------------+----------------------------------------+----------

1 row in set (0.00 sec)

mysql> **SELECT** **ENGINE\_LOCK\_ID** **as** **Lock\_Id,**

**ENGINE\_TRANSACTION\_ID** **as** **Trx\_id,**

**OBJECT\_NAME** **as** **`Table`,**

**INDEX\_NAME** **as** **`Index`,**

**LOCK\_DATA** **as** **Data,**

**LOCK\_MODE** **as** **Mode,**

**LOCK\_STATUS** **as** **Status,**

**LOCK\_TYPE** **as** **Type**

**FROM** **performance\_schema** **.data\_locks;**

+----------------------------------------+-----------------+---------+---------+------------+-------------

| Lock\_Id | Trx\_Id | Table | Index | Data | Mode

+----------------------------------------+-----------------+---------+---------+------------+-------------

|

|

|

|

|

|

|

|

|

|

|

Animals

Birds

139816129437696:1187:139816016603896

139816129437696:1188:139816016603808

43260

43260

NULL

NULL

NULL

NULL

IX

IS

|

| 139816129437696:28:4:2:139816016600896 | 43260 | Birds | PRIMARY | 'Buzzard' | S,REC\_NOT

| 139816129437696:27:4:2:139816016601240 | 43260 | Animals | PRIMARY | 'Aardvark' | X,REC\_NOT

| 139816129436888:1187:139816016597712 | 421291106147544 | Animals | NULL | NULL | IS

| 139816129436888:27:4:2:139816016594720 | 421291106147544 | Animals | PRIMARY | 'Aardvark' | S,REC\_NOT

+----------------------------------------+-----------------+---------+---------+------------+----------

6 rows in set (0.00 sec)

InnoDB only uses sequential transaction ids when a transaction attempts to modify the database. Thererfore, the previous read-only transaction id changes from 421291106148352 to 43260.

If client A attempts to update a row in Birds at the same time, this will lead to a deadlock:

mysql> **UPDATE** **Birds** **SET** **value=40** **WHERE** **name='Buzzard';**

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

InnoDB rolls back the transaction that caused the deadlock. The first update, from Client B, can now proceed.

The Information Schema contains the number of deadlocks:

mysql> **SELECT** **`count`** **FROM** **INFORMATION\_SCHEMA.INNODB\_METRICS**

**WHERE** **NAME="lock\_deadlocks";**

+-------+

| count |

+-------+

| 1 |

+-------+

1 row in set (0.00 sec)

The InnoDB status contains the following information about the deadlock and transactions. It also shows that the read-only transaction id 421291106147544 changes to sequential transaction id 43261.

mysql> **SHOW** **ENGINE** **INNODB** **STATUS;**

------------------------

LATEST DETECTED DEADLOCK

------------------------

2022-11-25 15:58:22 139815661168384

\*\*\* (1) TRANSACTION:

TRANSACTION 43260, ACTIVE 186 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)

MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2 updating

UPDATE Animals SET value=30 WHERE name='Aardvark'

\*\*\* (1) HOLDS THE LOCK(S):

RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.`Birds` trx id 43260 lock mo

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 7; hex 42757a7a617264; asc Buzzard;;

1: len 6; hex 00000000a8fb; asc ;;

2: len 7; hex 82000000e40110; asc ;;

3: len 4; hex 80000014; asc ;;

\*\*\* (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.`Animals` trx id 43260 lock\_

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 8; hex 416172647661726b; asc Aardvark;;

1: len 6; hex 00000000a8f9; asc ;;

2: len 7; hex 82000000e20110; asc ;;

3: len 4; hex 8000000a; asc ;;

\*\*\* (2) TRANSACTION:

TRANSACTION 43261, ACTIVE 209 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)

MySQL thread id 18, OS thread handle 139815618148096, query id 146 localhost u1 updating

UPDATE Birds SET value=40 WHERE name='Buzzard'

\*\*\* (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.`Animals` trx id 43261 lock

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 8; hex 416172647661726b; asc Aardvark;;

1: len 6; hex 00000000a8f9; asc ;;

2: len 7; hex 82000000e20110; asc ;;

3: len 4; hex 8000000a; asc ;;

\*\*\* (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.`Birds` trx id 43261 lock\_mode

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 7; hex 42757a7a617264; asc Buzzard;;

1: len 6; hex 00000000a8fb; asc ;;

2: len 7; hex 82000000e40110; asc ;;

3: len 4; hex 80000014; asc ;;

\*\*\* WE ROLL BACK TRANSACTION (2)

------------

TRANSACTIONS

------------

Trx id counter 43262

Purge done for trx's n:o < 43256 undo n:o < 0 state: running but idle

History list length 0

LIST OF TRANSACTIONS FOR EACH SESSION:

---TRANSACTION 421291106147544, not started

0 lock struct(s), heap size 1128, 0 row lock(s)

---TRANSACTION 421291106146736, not started

0 lock struct(s), heap size 1128, 0 row lock(s)

---TRANSACTION 421291106145928, not started

0 lock struct(s), heap size 1128, 0 row lock(s)

---TRANSACTION 43260, ACTIVE 219 sec

4 lock struct(s), heap size 1128, 2 row lock(s), undo log entries 1

MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2

The error log contains this information about transactions and locks:

mysql> **SELECT** **@@log\_error;**

+---------------------+

| @@log\_error |

+---------------------+

| /var/log/mysqld .log |

+ +

1 row in set (0.00 sec)

TRANSACTION 43260, ACTIVE 186 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)

MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2 updating

UPDATE Animals SET value=30 WHERE name='Aardvark'

RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.`Birds` trx id 43260 lock mode

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 7; hex 42757a7a617264; asc Buzzard;;

1: len 6; hex 00000000a8fb; asc ;;

2: len 7; hex 82000000e40110; asc ;;

3: len 4; hex 80000014; asc ;;

RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.`Animals` trx id 43260 lock\_mod

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 8; hex 416172647661726b; asc Aardvark;;

1: len 6; hex 00000000a8f9; asc ;;

2: len 7; hex 82000000e20110; asc ;;

3: len 4; hex 8000000a; asc ;;

TRANSACTION 43261, ACTIVE 209 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)

MySQL thread id 18, OS thread handle 139815618148096, query id 146 localhost u1 updating

UPDATE Birds SET value=40 WHERE name='Buzzard'

RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.`Animals` trx id 43261 lock mod

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 8; hex 416172647661726b; asc Aardvark;;

1: len 6; hex 00000000a8f9; asc ;;

2: len 7; hex 82000000e20110; asc ;;

3: len 4; hex 8000000a; asc ;;

RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.`Birds` trx id 43261 lock\_mo

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 4; compact format; info bits 0

0: len 7; hex 42757a7a617264; asc Buzzard;;

1: len 6; hex 00000000a8fb; asc ;;

2: len 7; hex 82000000e40110; asc ;;

3: len 4; hex 80000014; asc ;;

**15.7.5.2** **Deadlock** **Detection**

When deadlock detection is enabled (the default), InnoDB automatically detects transaction deadlocks and rolls back a transaction or transactions to break the deadlock. InnoDB tries to pick small transactions to roll back, where the size of a transaction is determined by the number of rows inserted, updated, or deleted.

InnoDB is aware of table locks if innodb\_table\_locks = 1 (the default) and autocommit = 0, and the MySQL layer above it knows about row-level locks. Otherwise, InnoDB cannot detect deadlocks where a table lock set by a MySQL LOCK TABLES statement or a lock set by a storage engine other than InnoDB is involved. Resolve these situations by setting the value of the innodb\_lock\_wait\_timeout system variable.

If the LATEST DETECTED DEADLOCK section of InnoDB Monitor output includes a message stating

TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH, WE WILL ROLL BACK

FOLLOWING TRANSACTION, this indicates that the number of transactions on the wait-for list has reached a limit of 200. A wait-for list that exceeds 200 transactions is treated as a deadlock and the transaction attempting to check the wait-for list is rolled back. The same error may also occur if the locking thread must look at more than 1,000,000 locks owned by transactions on the wait-for list.

For techniques to organize database operations to avoid deadlocks, see [Section 15.7.5, “Deadlocks in](#_bookmark4) [InnoDB”](#_bookmark4) .

**Disabling** **Deadlock** **Detection**

On high concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock. At times, it may be more efficient to disable deadlock detection and rely on the innodb\_lock\_wait\_timeout setting for transaction rollback when a deadlock occurs. Deadlock detection can be disabled using the innodb\_deadlock\_detect variable.

**15.7.5.3** **How** **to** **Minimize** **and** **Handle** **Deadlocks**

This section builds on the conceptual information about deadlocks in [Section 15.7.5.2, “Deadlock](#_bookmark3) [Detection”](#_bookmark3) . It explains how to organize database operations to minimize deadlocks and the subsequent error handling required in applications.

Deadlocks are a classic problem in transactional databases, but they are not dangerous unless

they are so frequent that you cannot run certain transactions at all. Normally, you must write your applications so that they are always prepared to re-issue a transaction if it gets rolled back because of a deadlock.

InnoDB uses automatic row-level locking. You can get deadlocks even in the case of transactions that just insert or delete a single row. That is because these operations are not really “atomic”; they automatically set locks on the (possibly several) index records of the row inserted or deleted.

You can cope with deadlocks and reduce the likelihood of their occurrence with the following techniques:

• At any time, issue SHOW ENGINE INNODB STATUS to determine the cause of the most recent deadlock. That can help you to tune your application to avoid deadlocks.

• If frequent deadlock warnings cause concern, collect more extensive debugging information by enabling the innodb\_print\_all\_deadlocks variable. Information about each deadlock, not

just the latest one, is recorded in the MySQL error log. Disable this option when you are finished debugging.

• Always be prepared to re-issue a transaction if it fails due to deadlock. Deadlocks are not dangerous. Just try again.

• Keep transactions small and short in duration to make them less prone to collision.

• Commit transactions immediately after making a set of related changes to make them less prone to collision. In particular, do not leave an interactive mysql session open for a long time with an uncommitted transaction.

• If you use locking reads (SELECT ... FOR UPDATE or SELECT ... FOR SHARE), try using a lower isolation level such as READ COMMITTED.

• When modifying multiple tables within a transaction, or different sets of rows in the same table, do those operations in a consistent order each time. Then transactions form well-defined queues and do not deadlock. For example, organize database operations into functions within your application, or call stored routines, rather than coding multiple similar sequences of INSERT, UPDATE, and DELETE

statements in different places.

• Add well-chosen indexes to your tables so that your queries scan fewer index records and set fewer locks. Use EXPLAIN SELECT to determine which indexes the MySQL server regards as the most appropriate for your queries.

• Use less locking. If you can afford to permit a SELECT to return data from an old snapshot, do not add a FOR UPDATE or FOR SHARE clause to it. Using the READ COMMITTED isolation level is good here, because each consistent read within the same transaction reads from its own fresh snapshot.

• If nothing else helps, serialize your transactions with table-level locks. The correct way to use

LOCK TABLES with transactional tables, such as InnoDB tables, is to begin a transaction with SET autocommit = 0 (not START TRANSACTION) followed by LOCK TABLES, and to not call UNLOCK TABLES until you commit the transaction explicitly. For example, if you need to write to table t1 and read from table t2, you can do this:

SET autocommit=0;

LOCK TABLES t1 WRITE, t2 READ, ...;

*...* *do* *something* *with* *tables* *t1* *and* *t2* *here* *...*

COMMIT;

UNLOCK TABLES;

Table-level locks prevent concurrent updates to the table, avoiding deadlocks at the expense of less responsiveness for a busy system.

• Another way to serialize transactions is to create an auxiliary “semaphore” table that contains just a single row. Have each transaction update that row before accessing other tables. In that way, all transactions happen in a serial fashion. Note that the InnoDB instant deadlock detection algorithm also works in this case, because the serializing lock is a row-level lock. With MySQL table-level locks, the timeout method must be used to resolve deadlocks.

**15.7.6** **Transaction** **Scheduling**

InnoDB uses the Contention-Aware Transaction Scheduling (CATS) algorithm to prioritize transactions that are waiting for locks. When multiple transactions are waiting for a lock on the same object, the CATS algorithm determines which transaction receives the lock first.

The CATS algorithm prioritizes waiting transactions by assigning a scheduling weight, which is computed based on the number of transactions that a transaction blocks. For example, if two transactions are waiting for a lock on the same object, the transaction that blocks the most transactions is assigned a greater scheduling weight. If weights are equal, priority is given to the longest waiting transaction.



**Note**

Prior to MySQL 8.0.20, InnoDB also uses a First In First Out (FIFO) algorithm to schedule transactions, and the CATS algorithm is used under heavy lock contention only. CATS algorithm enhancements in MySQL 8.0.20 rendered the FIFO algorithm redundant, permitting its removal. Transaction scheduling previously performed by the FIFO algorithm is performed by the CATS algorithm as of MySQL 8.0.20. In some cases, this change may affect the order in which transactions are granted locks.

You can view transaction scheduling weights by querying the TRX\_SCHEDULE\_WEIGHT column in the Information Schema INNODB\_TRX table. Weights are computed for waiting transactions only. Waiting transactions are those in a LOCK WAIT transaction execution state, as reported by the TRX\_STATE column. A transaction that is not waiting for a lock reports a NULL TRX\_SCHEDULE\_WEIGHT value.

INNODB\_METRICS counters are provided for monitoring of code-level transaction scheduling events. For information about using INNODB\_METRICS counters, see Section 15.15.6, “InnoDB INFORMATION\_SCHEMA Metrics Table” .

• lock\_rec\_release\_attempts

The number of attempts to release record locks. A single attempt may lead to zero or more record locks being released, as there may be zero or more record locks in a single structure.

• lock\_rec\_grant\_attempts

The number of attempts to grant record locks. A single attempt may result in zero or more record locks being granted.

• lock\_schedule\_refreshes

The number of times the wait-for graph was analyzed to update the scheduled transaction weights.

**15.8** **InnoDB** **Configuration**

This section provides configuration information and procedures for InnoDB initialization, startup, and various components and features of the InnoDB storage engine. For information about optimizing database operations for InnoDB tables, see Section 8.5, “Optimizing for InnoDB Tables” .

**15.8.1** **InnoDB** **Startup** **Configuration**

The first decisions to make about InnoDB configuration involve the configuration of data files, log files, page size, and memory buffers, which should be configured before initializing InnoDB. Modifying the configuration after InnoDB is initialized may involve non-trivial procedures.

This section provides information about specifying InnoDB settings in a configuration file, viewing InnoDB initialization information, and important storage considerations.

• [Specifying Options in a MySQL Option File](#_bookmark7)

• [Viewing InnoDB Initialization Information](#_bookmark8)

• [Important Storage Considerations](#_bookmark9)

• [System Tablespace Data File Configuration](#_bookmark10)

• [InnoDB Doublewrite Buffer File Configuration](#_bookmark11)

• [Redo Log Configuration](#_bookmark12)

• [Undo Tablespace Configuration](#_bookmark13)



• [Global Temporary Tablespace Configuration](#_bookmark14)

• [Session Temporary Tablespace Configuration](#_bookmark15)

• [Page Size Configuration](#_bookmark16)

• [Memory Configuration](#_bookmark17)

**Specifying** **Options** **in** **a** **MySQL** **Option** **File**

Because MySQL uses data file, log file, and page size settings to initialize InnoDB, it is recommended that you define these settings in an option file that MySQL reads at startup, prior to initializing InnoDB. Normally, InnoDB is initialized when the MySQL server is started for the first time.

You can place InnoDB options in the [mysqld] group of any option file that your server reads when it starts. The locations of MySQL option files are described in Section 4.2.2.2, “Using Option Files” .

To make sure that mysqld reads options only from a specific file (and mysqld-auto.cnf), use the --defaults-file option as the first option on the command line when starting the server:

mysqld --defaults-file=*path\_to\_option\_file*

**Viewing** **InnoDB** **Initialization** **Information**

To view InnoDB initialization information during startup, start mysqld from a command prompt, which prints initialization information to the console.

For example, on Windows, if mysqld is located in C:\Program Files\MySQL\MySQL Server 8.0\bin, start the MySQL server like this:

C:\> **"C:\Program** **Files\MySQL\MySQL** **Server** **8.0\bin\mysqld"** **--console**

On Unix-like systems, mysqld is located in the bin directory of your MySQL installation: $> **bin/mysqld** **--user=mysql** **&**

If you do not send server output to the console, check the error log after startup to see the initialization information InnoDB printed during the startup process.

For information about starting MySQL using other methods, see Section 2.9.5, “Starting and Stopping MySQL Automatically” .

**Note**

InnoDB does not open all user tables and associated data files at startup. However, InnoDB does check for the existence of tablespace files referenced in the data dictionary. If a tablespace file is not found, InnoDB logs an error and continues the startup sequence. Tablespace files referenced in the redo log may be opened during crash recovery for redo application.

**Important** **Storage** **Considerations**

Review the following storage-related considerations before proceeding with your startup configuration.

• In some cases, you can improve database performance by placing data and log files on separate physical disks. You can also use raw disk partitions (raw devices) for InnoDB data files, which may speed up I/O. See Using Raw Disk Partitions for the System Tablespace.

• InnoDB is a transaction-safe (ACID compliant) storage engine with commit, rollback, and crash- recovery capabilities to protect user data. **However,** **it** **cannot** **do** **so** if the underlying operating system or hardware does not work as advertised. Many operating systems or disk subsystems may delay or reorder write operations to improve performance. On some operating systems, the very fsync() system call that should wait until all unwritten data for a file has been flushed might actually return before the data has been flushed to stable storage. Because of this, an operating system

crash or a power outage may destroy recently committed data, or in the worst case, even corrupt the database because write operations have been reordered. If data integrity is important to you, perform “pull-the-plug” tests before using anything in production. On macOS, InnoDB uses a special fcntl() file flush method. Under Linux, it is advisable to **disable** **the** **write-back** **cache**.

On ATA/SATA disk drives, a command such hdparm -W0 /dev/hda may work to disable the write-back cache. **Beware** **that** **some** **drives** **or** **disk** **controllers** **may** **be** **unable** **to** **disable** **the** **write-back** **cache.**

• With regard to InnoDB recovery capabilities that protect user data, InnoDB uses a file flush technique involving a structure called the doublewrite buffer, which is enabled by default (innodb\_doublewrite=ON). The doublewrite buffer adds safety to recovery following an unexpected exit or power outage, and improves performance on most varieties of Unix by reducing the need for fsync() operations. It is recommended that the innodb\_doublewrite option remains enabled if you are concerned with data integrity or possible failures. For information about the doublewrite buffer, see Section 15.11.1, “InnoDB Disk I/O” .

• Before using NFS with InnoDB, review potential issues outlined in Using NFS with MySQL.

**System** **Tablespace** **Data** **File** **Configuration**

The innodb\_data\_file\_path option defines the name, size, and attributes of InnoDB system tablespace data files. If you do not configure this option prior to initializing the MySQL server, the default behavior is to create a single auto-extending data file, slightly larger than 12MB, named ibdata1:

mysql> **SHOW** **VARIABLES** **LIKE** **'innodb\_data\_file\_path';**

+-----------------------+------------------------+

| Variable\_name | Value |

+-----------------------+------------------------+

| innodb\_data\_file\_path | ibdata1:12M:autoextend |

+-----------------------+------------------------+

The full data file specification syntax includes the file name, file size, autoextend attribute, and max attribute:

*file\_name*:*file\_size* [:autoextend[:max:*max\_file\_size*]]

File sizes are specified in kilobytes, megabytes, or gigabytes by appending K, M or G to the size value. If specifying the data file size in kilobytes, do so in multiples of 1024. Otherwise, kilobyte values are rounded to nearest megabyte (MB) boundary. The sum of file sizes must be, at a minimum, slightly larger than 12MB.

You can specify more than one data file using a semicolon-separated list. For example:

[mysqld]

innodb\_data\_file\_path=ibdata1:50M;ibdata2:50M:autoextend

The autoextend and max attributes can be used only for the data file that is specified last.

When the autoextend attribute is specified, the data file automatically increases in size by 64MB increments as space is required. The innodb\_autoextend\_increment variable controls the increment size.

To specify a maximum size for an auto-extending data file, use the max attribute following the autoextend attribute. Use the max attribute only in cases where constraining disk usage is of critical importance. The following configuration permits ibdata1 to grow to a limit of 500MB:

[mysqld]

innodb\_data\_file\_path=ibdata1:12M:autoextend:max:500M

A minimum file size is enforced for the *first* system tablespace data file to ensure that there is enough

space for doublewrite buffer pages. The following table shows minimum file sizes for each InnoDB page size. The default InnoDB page size is 16384 (16KB).

|  |  |
| --- | --- |
| **Page** **Size** **(innodb\_page\_size)** | **Minimum** **File** **Size** |
| 16384 (16KB) or less | 3MB |
| 32768 (32KB) | 6MB |
| 65536 (64KB) | 12MB |

If your disk becomes full, you can add a data file on another disk. For instructions, see Resizing the System Tablespace.

The size limit for individual files is determined by your operating system. You can set the file size to more than 4GB on operating systems that support large files. You can also use raw disk partitions as data files. See Using Raw Disk Partitions for the System Tablespace.

InnoDB is not aware of the file system maximum file size, so be cautious on file systems where the maximum file size is a small value such as 2GB.

System tablespace files are created in the data directory by default (datadir). To specify an alternate location, use the innodb\_data\_home\_dir option. For example, to create a system tablespace data file in a directory named myibdata, use this configuration:

[mysqld]

innodb\_data\_home\_dir = /myibdata/

innodb\_data\_file\_path=ibdata1:50M:autoextend

A trailing slash is required when specifying a value for innodb\_data\_home\_dir. InnoDB does not create directories, so ensure that the specified directory exists before you start the server. Also, ensure sure that the MySQL server has the proper access rights to create files in the directory.

InnoDB forms the directory path for each data file by textually concatenating the value of innodb\_data\_home\_dir to the data file name. If innodb\_data\_home\_dir is not defined, the default value is “ ./” , which is the data directory. (The MySQL server changes its current working directory to the data directory when it begins executing.)

Alternatively, you can specify an absolute path for system tablespace data files. The following configuration is equivalent to the preceding one:

[mysqld]

innodb\_data\_file\_path=/myibdata/ibdata1:50M:autoextend

When you specify an absolute path for innodb\_data\_file\_path, the setting is not concatenated with the innodb\_data\_home\_dir setting. System tablespace files are created in the specified absolute path. The specified directory must exist before you start the server.

**InnoDB** **Doublewrite** **Buffer** **File** **Configuration**

As of MySQL 8.0.20, the doublewrite buffer storage area resides in doublewrite files, which provides flexibility with respect to the storage location of doublewrite pages. In previous releases, the doublewrite buffer storage area resided in the system tablespace. The innodb\_doublewrite\_dir variable defines the directory where InnoDB creates doublewrite files at startup. If no directory is specified, doublewrite files are created in the innodb\_data\_home\_dir directory, which defaults to the data directory if unspecified.

To have doublewrite files created in a location other than the innodb\_data\_home\_dir directory, configure innodb\_doublewrite\_dir variable. For example:

innodb\_doublewrite\_dir=*/path/to/doublewrite\_directory*

Other doublewrite buffer variables permit defining the number of doublewrite files, the number of pages per thread, and the doublewrite batch size. For more information about doublewrite buffer configuration, see Section 15.6.4, “Doublewrite Buffer” .



**Redo** **Log** **Configuration**

From MySQL 8.0.30, the amount of disk space occupied by redo log files is controlled by the innodb\_redo\_log\_capacity variable, which can be set at startup or runtime; for example, to set the variable to 8GB in an option file, add the following entry:

[mysqld]

innodb\_redo\_log\_capacity = 8589934592

For information about configuring redo log capacity at runtime, see Configuring Redo Log Capacity (MySQL 8.0.30 or Higher).

The innodb\_redo\_log\_capacity variable supersedes the innodb\_log\_file\_size and innodb\_log\_files\_in\_group variables, which are deprecated. When the innodb\_redo\_log\_capacity setting is defined, the innodb\_log\_file\_size and innodb\_log\_files\_in\_group settings are ignored; otherwise, these settings are used to compute the innodb\_redo\_log\_capacity setting (innodb\_log\_files\_in\_group \* innodb\_log\_file\_size = innodb\_redo\_log\_capacity). If none of those variables are set, innodb\_redo\_log\_capacity is set to the default value, which is 104857600 bytes (100MB). The maximum setting is 128GB.

From MySQL 8.0.30, InnoDB attempts to maintain 32 redo log files, with each file equal to 1/32 \* innodb\_redo\_log\_capacity. The redo log files reside in the #innodb\_redo directory in the data directory unless a different directory was specified by the innodb\_log\_group\_home\_dir variable. If innodb\_log\_group\_home\_dir was defined, the redo log files reside in the #innodb\_redo directory in that directory. For more information, see Section 15.6.5, “Redo Log” .

Before MySQL 8.0.30, InnoDB creates two 5MB redo log files named ib\_logfile0 and ib\_logfile1 in the data directory by default. You can define a different number of redo log files and different redo log file size when initializing the MySQL Server instance by configuring the innodb\_log\_files\_in\_group and innodb\_log\_file\_size variables.

• innodb\_log\_files\_in\_group defines the number of log files in the log group. The default and recommended value is 2.

• innodb\_log\_file\_size defines the size in bytes of each log file in the log group. The combined log file size (innodb\_log\_file\_size \* innodb\_log\_files\_in\_group) cannot exceed the maximum value, which is slightly less than 512GB. A pair of 255 GB log files, for example, approaches the limit but does not exceed it. The default log file size is 48MB. Generally, the combined size of the log files should be large enough that the server can smooth out peaks and troughs in workload activity, which often means that there is enough redo log space to handle more than an hour of write activity. A larger log file size means less checkpoint flush activity in the buffer pool, which reduces disk I/O. For additional information, see Section 8.5.4, “Optimizing InnoDB Redo Logging” .

The innodb\_log\_group\_home\_dir defines directory path to the InnoDB log files. You might use this option to place InnoDB redo log files in a different physical storage location than InnoDB data files to avoid potential I/O resource conflicts; for example:

[mysqld]

innodb\_log\_group\_home\_dir = /dr3/iblogs

**Note**

InnoDB does not create directories, so make sure that the log directory exists before you start the server. Use the Unix or DOS mkdir command to create any necessary directories.

Make sure that the MySQL server has the proper access rights to create files in the log directory. More generally, the server must have access rights in any directory where it needs to create files.

**Undo** **Tablespace** **Configuration**

Undo logs, by default, reside in two undo tablespaces created when the MySQL instance is initialized.

The innodb\_undo\_directory variable defines the path where InnoDB creates default undo tablespaces. If that variable is undefined, default undo tablespaces are created in the data directory.

The innodb\_undo\_directory variable is not dynamic. Configuring it requires restarting the server. The I/O patterns for undo logs make undo tablespaces good candidates for SSD storage.

For information about configuring additional undo tablespaces, see Section 15.6.3.4, “Undo Tablespaces” .

**Global** **Temporary** **Tablespace** **Configuration**

The global temporary tablespace stores rollback segments for changes made to user-created temporary tables.

A single auto-extending global temporary tablespace data file named ibtmp1 in the innodb\_data\_home\_dir directory by default. The initial file size is slightly larger than 12MB.

The innodb\_temp\_data\_file\_path option specifies the path, file name, and file size for global temporary tablespace data files. File size is specified in KB, MB, or GB by appending K, M, or G to the size value. The file size or combined file size must be slightly larger than 12MB.

To specify an alternate location for global temporary tablespace data files, configure the innodb\_temp\_data\_file\_path option at startup.

**Session** **Temporary** **Tablespace** **Configuration**

In MySQL 8.0.15 and earlier, session temporary tablespaces store user-created temporary tables and internal temporary tables created by the optimizer when InnoDB is configured as the on-disk storage engine for internal temporary tables (internal\_tmp\_disk\_storage\_engine=InnoDB). From MySQL 8.0.16, InnoDB is always used as the on-disk storage engine for internal temporary tables.

The innodb\_temp\_tablespaces\_dir variable defines the location where InnoDB creates session temporary tablespaces. The default location is the #innodb\_temp directory in the data directory.

To specify an alternate location for session temporary tablespaces, configure the innodb\_temp\_tablespaces\_dir variable at startup. A fully qualified path or path relative to the data directory is permitted.

**Page** **Size** **Configuration**

The innodb\_page\_size option specifies the page size for all InnoDB tablespaces in a MySQL instance. This value is set when the instance is created and remains constant afterward. Valid values are 64KB, 32KB, 16KB (the default), 8KB, and 4KB. Alternatively, you can specify page size in bytes (65536, 32768, 16384, 8192, 4096).

The default 16KB page size is appropriate for a wide range of workloads, particularly for queries involving table scans and DML operations involving bulk updates. Smaller page sizes might be more efficient for OLTP workloads involving many small writes, where contention can be an issue when a single page contains many rows. Smaller pages can also be more efficient for SSD storage devices, which typically use small block sizes. Keeping the InnoDB page size close to the storage device block size minimizes the amount of unchanged data that is rewritten to disk.

**Memory** **Configuration**

MySQL allocates memory to various caches and buffers to improve performance of database operations. When allocating memory for InnoDB, always consider memory required by the operating system, memory allocated to other applications, and memory allocated for other MySQL buffers and caches. For example, if you use MyISAM tables, consider the amount of memory allocated for the key



buffer (key\_buffer\_size). For an overview of MySQL buffers and caches, see Section 8.12.3.1, “How MySQL Uses Memory” .

Buffers specific to InnoDB are configured using the following parameters:

• innodb\_buffer\_pool\_size defines size of the buffer pool, which is the memory area that holds cached data for InnoDB tables, indexes, and other auxiliary buffers. The size of

the buffer pool is important for system performance, and it is typically recommended that innodb\_buffer\_pool\_size is configured to 50 to 75 percent of system memory. The default buffer pool size is 128MB. For additional guidance, see Section 8. 12.3. 1, “How MySQL Uses Memory” . For information about how to configure InnoDB buffer pool size, see [Section 15.8.3.1,](#_bookmark18) [“Configuring InnoDB Buffer Pool Size”](#_bookmark18) . Buffer pool size can be configured at startup or dynamically.

On systems with a large amount of memory, you can improve concurrency by dividing the buffer pool into multiple buffer pool instances. The number of buffer pool instances is controlled by the by innodb\_buffer\_pool\_instances option. By default, InnoDB creates one buffer pool instance. The number of buffer pool instances can be configured at startup. For more information, see [Section 15.8.3.2, “Configuring Multiple Buffer Pool Instances”](#_bookmark19) .

• innodb\_log\_buffer\_size defines the size of the buffer that InnoDB uses to write to the log files on disk. The default size is 16MB. A large log buffer enables large transactions to run

without writing the log to disk before the transactions commit. If you have transactions that update, insert, or delete many rows, you might consider increasing the size of the log buffer to save disk I/O. innodb\_log\_buffer\_size can be configured at startup. For related information, see Section 8.5.4, “Optimizing InnoDB Redo Logging” .

**Warning**

On 32-bit GNU/Linux x86, if memory usage is set too high, glibc may permit the process heap to grow over the thread stacks, causing a server failure. It is a risk if the memory allocated to the mysqld process for global and per-thread buffers and caches is close to or exceeds 2GB.

A formula similar to the following that calculates global and per-thread memory allocation for MySQL can be used to estimate MySQL memory usage. You may need to modify the formula to account for buffers and caches in your MySQL version and configuration. For an overview of MySQL buffers and caches, see Section 8.12.3.1, “How MySQL Uses Memory” .

innodb\_buffer\_pool\_size

+ key\_buffer\_size

+ max\_connections\*(sort\_buffer\_size+read\_buffer\_size+binlog\_cache\_size)

+ max\_connections\*2MB

Each thread uses a stack (often 2MB, but only 256KB in MySQL binaries provided by Oracle Corporation.) and in the worst case also uses sort\_buffer\_size + read\_buffer\_size additional memory.

On Linux, if the kernel is enabled for large page support, InnoDB can use large pages to allocate memory for its buffer pool. See Section 8.12.3.3, “Enabling Large Page Support” .

**15.8.2** **Configuring** **InnoDB** **for** **Read-Only** **Operation**

You can query InnoDB tables where the MySQL data directory is on read-only media by enabling the --innodb-read-only configuration option at server startup.

**How** **to** **Enable**

To prepare an instance for read-only operation, make sure all the necessary information is flushed to the data files before storing it on the read-only medium. Run the server with change buffering disabled (innodb\_change\_buffering=0) and do a slow shutdown.



To enable read-only mode for an entire MySQL instance, specify the following configuration options at server startup:

• --innodb-read-only=1

• If the instance is on read-only media such as a DVD or CD, or the /var directory is not writeable by all: --pid-file=*path\_on\_writeable\_media* and --event-scheduler=disabled

• --innodb-temp-data-file-path. This option specifies the path, file name, and file size for InnoDB temporary tablespace data files. The default setting is ibtmp1:12M:autoextend, which creates the ibtmp1 temporary tablespace data file in the data directory. To prepare an instance for read-only operation, set innodb\_temp\_data\_file\_path to a location outside of the data directory. The path must be relative to the data directory. For example:

--innodb-temp-data-file-path=../../../tmp/ibtmp1:12M:autoextend

As of MySQL 8.0, enabling innodb\_read\_only prevents table creation and drop operations for all storage engines. These operations modify data dictionary tables in the mysql system database, but those tables use the InnoDB storage engine and cannot be modified when innodb\_read\_only is enabled. The same restriction applies to any operation that modifies data dictionary tables, such as ANALYZE TABLE and ALTER TABLE *tbl\_name* ENGINE=*engine\_name*.

In addition, other tables in the mysql system database use the InnoDB storage engine in MySQL 8.0. Making those tables read only results in restrictions on operations that modify them. For example, CREATE USER, GRANT, REVOKE, and INSTALL PLUGIN operations are not permitted in read-only mode.

**Usage** **Scenarios**

This mode of operation is appropriate in situations such as:

• Distributing a MySQL application, or a set of MySQL data, on a read-only storage medium such as a DVD or CD.

• Multiple MySQL instances querying the same data directory simultaneously, typically in a data warehousing configuration. You might use this technique to avoid bottlenecks that can occur with a heavily loaded MySQL instance, or you might use different configuration options for the various instances to tune each one for particular kinds of queries.

• Querying data that has been put into a read-only state for security or data integrity reasons, such as archived backup data.

**Note**

This feature is mainly intended for flexibility in distribution and deployment, rather than raw performance based on the read-only aspect. See Section 8.5.3, “Optimizing InnoDB Read-Only Transactions” for ways to tune the performance of read-only queries, which do not require making the entire server read-only.

**How** **It** **Works**

When the server is run in read-only mode through the --innodb-read-only option, certain InnoDB features and components are reduced or turned off entirely:

• No change buffering is done, in particular no merges from the change buffer. To make sure the change buffer is empty when you prepare the instance for read-only operation, disable change buffering (innodb\_change\_buffering=0) and do a slow shutdown first.

• There is no crash recovery phase at startup. The instance must have performed a slow shutdown before being put into the read-only state.

• Because the redo log is not used in read-only operation, you can set innodb\_log\_file\_size to the smallest size possible (1 MB) before making the instance read-only.

• Most background threads are turned off. I/O read threads remain, as well as I/O write threads and a page flush coordinator thread for writes to temporary files, which are permitted in read-only mode. A buffer pool resize thread also remains active to enable online resizing of the buffer pool.

• Information about deadlocks, monitor output, and so on is not written to temporary files. As a consequence, SHOW ENGINE INNODB STATUS does not produce any output.

• Changes to configuration option settings that would normally change the behavior of write operations, have no effect when the server is in read-only mode.

• The MVCC processing to enforce isolation levels is turned off. All queries read the latest version of a record, because update and deletes are not possible.

• The undo log is not used. Disable any settings for the innodb\_undo\_tablespaces and innodb\_undo\_directory configuration options.

**15.8.3** **InnoDB** **Buffer** **Pool** **Configuration**

This section provides configuration and tuning information for the InnoDB buffer pool.

**15.8.3.1** **Configuring** **InnoDB** **Buffer** **Pool** **Size**

You can configure InnoDB buffer pool size offline or while the server is running. Behavior described in this section applies to both methods. For additional information about configuring buffer pool size online, see [Configuring InnoDB Buffer Pool Size Online](#_bookmark20).

When increasing or decreasing innodb\_buffer\_pool\_size, the operation is performed in chunks. Chunk size is defined by the innodb\_buffer\_pool\_chunk\_size configuration option, which has a default of 128M. For more information, see [Configuring InnoDB Buffer Pool Chunk Size](#_bookmark21).

Buffer pool size must always be equal to or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances. If you configure innodb\_buffer\_pool\_size

to a value that is not equal to or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances, buffer pool size is automatically adjusted to a value that is equal to or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances.

In the following example, innodb\_buffer\_pool\_size is set to 8G, and innodb\_buffer\_pool\_instances is set to 16. innodb\_buffer\_pool\_chunk\_size is 128M, which is the default value.

8G is a valid innodb\_buffer\_pool\_size value because 8G is a multiple of innodb\_buffer\_pool\_instances=16 \* innodb\_buffer\_pool\_chunk\_size=128M, which is

2G.

$> **mysqld** **--innodb-buffer-pool-size=8G** **--innodb-buffer-pool-instances=16**

mysql> **SELECT** **@@innodb\_buffer\_pool\_size/1024/1024/1024;**

+------------------------------------------+

| @@innodb\_buffer\_pool\_size/1024/1024/1024 |

+------------------------------------------+

|  |  |
| --- | --- |
| | | 8.000000000000 | |

+------------------------------------------+

In this example, innodb\_buffer\_pool\_size is set to 9G, and innodb\_buffer\_pool\_instances is set to 16. innodb\_buffer\_pool\_chunk\_size is 128M, which is the default

value. In this case, 9G is not a multiple of innodb\_buffer\_pool\_instances=16 \* innodb\_buffer\_pool\_chunk\_size=128M, so innodb\_buffer\_pool\_size is adjusted to 10G, which is a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances.

$> **mysqld** **--innodb-buffer-pool-size=9G** **--innodb-buffer-pool-instances=16**

mysql> **SELECT** **@@innodb\_buffer\_pool\_size/1024/1024/1024;**

+------------------------------------------+

| @@innodb\_buffer\_pool\_size/1024/1024/1024 |

+------------------------------------------+

| 10.000000000000 |

+------------------------------------------+

**Configuring** **InnoDB** **Buffer** **Pool** **Chunk** **Size**

innodb\_buffer\_pool\_chunk\_size can be increased or decreased in 1MB (1048576 byte) units

but can only be modified at startup, in a command line string or in a MySQL configuration file. Command line:

$> **mysqld** **--innodb-buffer-pool-chunk-size=134217728**

Configuration file:

[mysqld]

innodb\_buffer\_pool\_chunk\_size=134217728

The following conditions apply when altering innodb\_buffer\_pool\_chunk\_size:

• If the new innodb\_buffer\_pool\_chunk\_size value \* innodb\_buffer\_pool\_instances is larger than the current buffer pool size when the buffer pool is initialized, innodb\_buffer\_pool\_chunk\_size is truncated to innodb\_buffer\_pool\_size / innodb\_buffer\_pool\_instances.

For example, if the buffer pool is initialized with a size of 2GB (2147483648 bytes), 4 buffer pool instances, and a chunk size of 1GB (1073741824 bytes), chunk size is truncated to a value equal to innodb\_buffer\_pool\_size / innodb\_buffer\_pool\_instances, as shown below:

$> **mysqld** **--innodb-buffer-pool-size=2147483648** **--innodb-buffer-pool-instances=4**

**--innodb-buffer-pool-chunk-size=1073741824;**

mysql> **SELECT** **@@innodb\_buffer\_pool\_size;**

+---------------------------+

| @@innodb\_buffer\_pool\_size |

+---------------------------+

| 2147483648 |

+---------------------------+

mysql> **SELECT** **@@innodb\_buffer\_pool\_instances;**

+--------------------------------+

| @@innodb\_buffer\_pool\_instances |

+--------------------------------+

| 4 |

+--------------------------------+

# Chunk size was set to 1GB (1073741824 bytes) on startup but was

# truncated to innodb\_buffer\_pool\_size / innodb\_buffer\_pool\_instances

mysql> **SELECT** **@@innodb\_buffer\_pool\_chunk\_size;**

+---------------------------------+

| @@innodb\_buffer\_pool\_chunk\_size |

+---------------------------------+

| 536870912 |

+---------------------------------+

• Buffer pool size must always be equal to or a multiple of innodb\_buffer\_pool\_chunk\_size

\* innodb\_buffer\_pool\_instances. If you alter innodb\_buffer\_pool\_chunk\_size, innodb\_buffer\_pool\_size is automatically adjusted to a value that is equal to or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances. The adjustment occurs when the buffer pool is initialized. This behavior is demonstrated in the following example:

# The buffer pool has a default size of 128MB (134217728 bytes)

mysql> **SELECT** **@@innodb\_buffer\_pool\_size;**

+---------------------------+

| @@innodb\_buffer\_pool\_size |

+---------------------------+

| 134217728 |

+---------------------------+

# The chunk size is also 128MB (134217728 bytes)

mysql> **SELECT** **@@innodb\_buffer\_pool\_chunk\_size;**

+---------------------------------+

| @@innodb\_buffer\_pool\_chunk\_size |

+---------------------------------+

| 134217728 |

+---------------------------------+

# There is a single buffer pool instance

mysql> **SELECT** **@@innodb\_buffer\_pool\_instances;**

+--------------------------------+

| @@innodb\_buffer\_pool\_instances |

+--------------------------------+

| 1 |

+--------------------------------+

# Chunk size is decreased by 1MB (1048576 bytes) at startup

# (134217728 - 1048576 = 133169152):

$> **mysqld** **--innodb-buffer-pool-chunk-size=133169152**

mysql> **SELECT** **@@innodb\_buffer\_pool\_chunk\_size;**

+---------------------------------+

| @@innodb\_buffer\_pool\_chunk\_size |

+---------------------------------+

| 133169152 |

+---------------------------------+

# Buffer pool size increases from 134217728 to 266338304

# Buffer pool size is automatically adjusted to a value that is equal to

# or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances

mysql> **SELECT** **@@innodb\_buffer\_pool\_size;**

+---------------------------+

| @@innodb\_buffer\_pool\_size |

+---------------------------+

| 266338304 |

+---------------------------+

This example demonstrates the same behavior but with multiple buffer pool instances:

# The buffer pool has a default size of 2GB (2147483648 bytes)

mysql> **SELECT** **@@innodb\_buffer\_pool\_size;**

+---------------------------+

| @@innodb\_buffer\_pool\_size |

+---------------------------+

| 2147483648 |

+---------------------------+

# The chunk size is .5 GB (536870912 bytes)

mysql> **SELECT** **@@innodb\_buffer\_pool\_chunk\_size;**

+---------------------------------+

| @@innodb\_buffer\_pool\_chunk\_size |

+---------------------------------+

| 536870912 |

+---------------------------------+

# There are 4 buffer pool instances

mysql> **SELECT** **@@innodb\_buffer\_pool\_instances;**

+--------------------------------+

| @@innodb\_buffer\_pool\_instances |

+--------------------------------+

| 4 |

+--------------------------------+

# Chunk size is decreased by 1MB (1048576 bytes) at startup



# (536870912 - 1048576 = 535822336):

$> **mysqld** **--innodb-buffer-pool-chunk-size=535822336**

mysql> **SELECT** **@@innodb\_buffer\_pool\_chunk\_size;**

+---------------------------------+

| @@innodb\_buffer\_pool\_chunk\_size |

+---------------------------------+

| 535822336 |

+---------------------------------+

# Buffer pool size increases from 2147483648 to 4286578688

# Buffer pool size is automatically adjusted to a value that is equal to

# or a multiple of innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances

mysql> **SELECT** **@@innodb\_buffer\_pool\_size;**

+---------------------------+

| @@innodb\_buffer\_pool\_size |

+---------------------------+

| 4286578688 |

+---------------------------+

Care should be taken when changing innodb\_buffer\_pool\_chunk\_size, as changing this value can increase the size of the buffer pool, as shown in the examples above. Before you change innodb\_buffer\_pool\_chunk\_size, calculate the effect on innodb\_buffer\_pool\_size to ensure that the resulting buffer pool size is acceptable.

**Note**

To avoid potential performance issues, the number of chunks (innodb\_buffer\_pool\_size / innodb\_buffer\_pool\_chunk\_size) should not exceed 1000.

**Configuring** **InnoDB** **Buffer** **Pool** **Size** **Online**

The innodb\_buffer\_pool\_size configuration option can be set dynamically using a SET statement, allowing you to resize the buffer pool without restarting the server. For example:

mysql> **SET** **GLOBAL** **innodb\_buffer\_pool\_size=402653184;**

**Note**

The buffer pool size must be equal to or a multiple of

innodb\_buffer\_pool\_chunk\_size \* innodb\_buffer\_pool\_instances. Changing those variable settings requires restarting the server.

Active transactions and operations performed through InnoDB APIs should be completed before resizing the buffer pool. When initiating a resizing operation, the operation does not start until all active transactions are completed. Once the resizing operation is in progress, new transactions and operations that require access to the buffer pool must wait until the resizing operation finishes. The exception to the rule is that concurrent access to the buffer pool is permitted while the buffer pool is defragmented and pages are withdrawn when buffer pool size is decreased. A drawback of allowing concurrent access is that it could result in a temporary shortage of available pages while pages are being withdrawn.

**Note**

Nested transactions could fail if initiated after the buffer pool resizing operation begins.

**Monitoring** **Online** **Buffer** **Pool** **Resizing** **Progress**

The Innodb\_buffer\_pool\_resize\_status variable reports a string value indicating buffer pool resizing progress; for example:

mysql> **SHOW** **STATUS** **WHERE** **Variable\_name='InnoDB\_buffer\_pool\_resize\_status';**

+----------------------------------+----------------------------------+

| Variable\_name | Value |

+----------------------------------+----------------------------------+

| Innodb\_buffer\_pool\_resize\_status | Resizing also other hash tables . |

+----------------------------------+----------------------------------+

From MyQL 8.0.31, you can also monitor an online buffer pool resizing

operation using the Innodb\_buffer\_pool\_resize\_status\_code and Innodb\_buffer\_pool\_resize\_status\_progress status variables, which report numeric values, preferable for programmatic monitoring.

The Innodb\_buffer\_pool\_resize\_status\_code status variable reports a status code indicating the stage of an online buffer pool resizing operation. Status codes include:

• 0: No Resize operation in progress

• 1: Starting Resize

• 2: Disabling AHI (Adaptive Hash Index)

• 3: Withdrawing Blocks

• 4: Acquiring Global Lock

• 5: Resizing Pool

• 6: Resizing Hash

• 7: Resizing Failed

The Innodb\_buffer\_pool\_resize\_status\_progress status variable reports a percentage value indicating the progress of each stage. The percentage value is updated after each buffer pool instance is processed. As the status (reported by Innodb\_buffer\_pool\_resize\_status\_code) changes from one status to another, the percentage value is reset to 0.

The following query returns a string value indicating the buffer pool resizing progress, a code indicating the current stage of the operation, and the current progress of that stage, expressed as a percentage value:

SELECT variable\_name, variable\_value

FROM performance\_schema .global\_status

WHERE LOWER(variable\_name) LIKE "innodb\_buffer\_pool\_resize%";

Buffer pool resizing progress is also visible in the server error log. This example shows notes that are logged when increasing the size of the buffer pool:

[Note] InnoDB: Resizing buffer pool from 134217728 to 4294967296. (unit=134217728)

[Note] InnoDB: disabled adaptive hash index .

[Note] InnoDB: buffer pool 0 : 31 chunks (253952 blocks) was added .

[Note] InnoDB: buffer pool 0 : hash tables were resized .

[Note] InnoDB: Resized hash tables at lock\_sys, adaptive hash index, dictionary .

[Note] InnoDB: completed to resize buffer pool from 134217728 to 4294967296.

[Note] InnoDB: re-enabled adaptive hash index.

This example shows notes that are logged when decreasing the size of the buffer pool:

[Note] InnoDB: Resizing buffer pool from 4294967296 to 134217728. (unit=134217728)

[Note] InnoDB: disabled adaptive hash index .

[Note] InnoDB: buffer pool 0 : start to withdraw the last 253952 blocks .

[Note] InnoDB: buffer pool 0 : withdrew 253952 blocks from free list . tried to relocate

0 pages . (253952/253952)

[Note] InnoDB: buffer pool 0 : withdrawn target 253952 blocks .

[Note] InnoDB: buffer pool 0 : 31 chunks (253952 blocks) was freed .

[Note] InnoDB: buffer pool 0 : hash tables were resized .

[Note] InnoDB: Resized hash tables at lock\_sys, adaptive hash index, dictionary .

[Note] InnoDB: completed to resize buffer pool from 4294967296 to 134217728.

[Note] InnoDB: re-enabled adaptive hash index.

From MySQL 8.0.31, starting the server with --log-error-verbosity=3 logs additional information to the error log during an online buffer pool resizing operation. Additional information includes the status codes reported by Innodb\_buffer\_pool\_resize\_status\_code and the percentage progress value reported by Innodb\_buffer\_pool\_resize\_status\_progress.

[Note] [MY-012398] [InnoDB] Requested to resize buffer pool. (new size: 1073741824 bytes)

[Note] [MY-013954] [InnoDB] Status code 1: Resizing buffer pool from 134217728 to 1073741824

(unit=134217728).

[Note] [MY-013953] [InnoDB] Status code 1: 100% complete

[Note] [MY-013952] [InnoDB] Status code 1: Completed

[Note] [MY-013954] [InnoDB] Status code 2: Disabling adaptive hash index .

[Note] [MY-011885] [InnoDB] disabled adaptive hash index .

[Note] [MY-013953] [InnoDB] Status code 2: 100% complete

[Note] [MY-013952] [InnoDB] Status code 2: Completed

[Note] [MY-013954] [InnoDB] Status code 3: Withdrawing blocks to be shrunken .

[Note] [MY-013953] [InnoDB] Status code 3: 100% complete

[Note] [MY-013952] [InnoDB] Status code 3: Completed

[Note] [MY-013954] [InnoDB] Status code 4: Latching whole of buffer pool .

[Note] [MY-013953] [InnoDB] Status code 4: 14% complete

[Note] [MY-013953] [InnoDB] Status code 4: 28% complete

[Note] [MY-013953] [InnoDB] Status code 4: 42% complete

[Note] [MY-013953] [InnoDB] Status code 4: 57% complete

[Note] [MY-013953] [InnoDB] Status code 4: 71% complete

[Note] [MY-013953] [InnoDB] Status code 4: 85% complete

[Note] [MY-013953] [InnoDB] Status code 4: 100% complete

[Note] [MY-013952] [InnoDB] Status code 4: Completed

[Note] [MY-013954] [InnoDB] Status code 5: Starting pool resize

[Note] [MY-013954] [InnoDB] Status code 5: buffer pool 0 : resizing with chunks 1 to 8.

[Note] [MY-011891] [InnoDB] buffer pool 0 : 7 chunks (57339 blocks) were added .

[Note] [MY-013953] [InnoDB] Status code 5: 100% complete

[Note] [MY-013952] [InnoDB] Status code 5: Completed

[Note] [MY-013954] [InnoDB] Status code 6: Resizing hash tables .

[Note] [MY-011892] [InnoDB] buffer pool 0 : hash tables were resized .

[Note] [MY-013953] [InnoDB] Status code 6: 100% complete

[Note] [MY-013954] [InnoDB] Status code 6: Resizing also other hash tables .

[Note] [MY-011893] [InnoDB] Resized hash tables at lock\_sys, adaptive hash index, dictionary .

[Note] [MY-011894] [InnoDB] Completed to resize buffer pool from 134217728 to 1073741824.

[Note] [MY-011895] [InnoDB] Re-enabled adaptive hash index .

[Note] [MY-013952] [InnoDB] Status code 6: Completed

[Note] [MY-013954] [InnoDB] Status code 0: Completed resizing buffer pool at 220826 6:25:46 .

[Note] [MY-013953] [InnoDB] Status code 0: 100% complete

**Online** **Buffer** **Pool** **Resizing** **Internals**

The resizing operation is performed by a background thread. When increasing the size of the buffer pool, the resizing operation:

• Adds pages in chunks (chunk size is defined by innodb\_buffer\_pool\_chunk\_size)

• Converts hash tables, lists, and pointers to use new addresses in memory

• Adds new pages to the free list

While these operations are in progress, other threads are blocked from accessing the buffer pool. When decreasing the size of the buffer pool, the resizing operation:

• Defragments the buffer pool and withdraws (frees) pages

• Removes pages in chunks (chunk size is defined by innodb\_buffer\_pool\_chunk\_size)

• Converts hash tables, lists, and pointers to use new addresses in memory

Of these operations, only defragmenting the buffer pool and withdrawing pages allow other threads to access to the buffer pool concurrently.

**15.8.3.2** **Configuring** **Multiple** **Buffer** **Pool** **Instances**

For systems with buffer pools in the multi-gigabyte range, dividing the buffer pool into separate instances can improve concurrency, by reducing contention as different threads read and write to cached pages. This feature is typically intended for systems with a buffer pool size in the multi-gigabyte range. Multiple buffer pool instances are configured using the innodb\_buffer\_pool\_instances configuration option, and you might also adjust the innodb\_buffer\_pool\_size value.

When the InnoDB buffer pool is large, many data requests can be satisfied by retrieving from memory. You might encounter bottlenecks from multiple threads trying to access the buffer pool at once. You can enable multiple buffer pools to minimize this contention. Each page that is stored in or read from the buffer pool is assigned to one of the buffer pools randomly, using a hashing function. Each buffer pool manages its own free lists, flush lists, LRUs, and all other data structures connected to a buffer pool. Prior to MySQL 8.0, each buffer pool was protected by its own buffer pool mutex. In MySQL 8.0 and later, the buffer pool mutex was replaced by several list and hash protecting mutexes, to reduce contention.

To enable multiple buffer pool instances, set the innodb\_buffer\_pool\_instances configuration option to a value greater than 1 (the default) up to 64 (the maximum). This option takes effect

only when you set innodb\_buffer\_pool\_size to a size of 1GB or more. The total size you specify is divided among all the buffer pools. For best efficiency, specify a combination of innodb\_buffer\_pool\_instances and innodb\_buffer\_pool\_size so that each buffer pool instance is at least 1GB.

For information about modifying InnoDB buffer pool size, see [Section 15.8.3.1, “Configuring InnoDB](#_bookmark18) [Buffer Pool Size”](#_bookmark18) .

**15.8.3.3** **Making** **the** **Buffer** **Pool** **Scan** **Resistant**

Rather than using a strict LRU algorithm, InnoDB uses a technique to minimize the amount of data that is brought into the buffer pool and never accessed again. The goal is to make sure that frequently accessed (“hot”) pages remain in the buffer pool, even as read-ahead and full table scans bring in new blocks that might or might not be accessed afterward.

Newly read blocks are inserted into the middle of the LRU list. All newly read pages are inserted at a location that by default is 3/8 from the tail of the LRU list. The pages are moved to the front of the list (the most-recently used end) when they are accessed in the buffer pool for the first time. Thus, pages that are never accessed never make it to the front portion of the LRU list, and “age out” sooner than with a strict LRU approach. This arrangement divides the LRU list into two segments, where the pages downstream of the insertion point are considered “old” and are desirable victims for LRU eviction.

For an explanation of the inner workings of the InnoDB buffer pool and specifics about the LRU algorithm, see Section 15.5.1, “Buffer Pool” .

You can control the insertion point in the LRU list and choose whether InnoDB applies the same optimization to blocks brought into the buffer pool by table or index scans. The configuration parameter innodb\_old\_blocks\_pct controls the percentage of “old” blocks in the LRU list. The default value of innodb\_old\_blocks\_pct is 37, corresponding to the original fixed ratio of 3/8. The value range is 5 (new pages in the buffer pool age out very quickly) to 95 (only 5% of the buffer pool is reserved for hot pages, making the algorithm close to the familiar LRU strategy).

The optimization that keeps the buffer pool from being churned by read-ahead can avoid similar problems due to table or index scans. In these scans, a data page is typically accessed

a few times in quick succession and is never touched again. The configuration parameter innodb\_old\_blocks\_time specifies the time window (in milliseconds) after the first access to a page during which it can be accessed without being moved to the front (most-recently used end) of the LRU list. The default value of innodb\_old\_blocks\_time is 1000. Increasing this value makes more and more blocks likely to age out faster from the buffer pool.

Both innodb\_old\_blocks\_pct and innodb\_old\_blocks\_time can be specified in the MySQL option file (my.cnf or my.ini) or changed at runtime with the SET GLOBAL statement. Changing the value at runtime requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

To help you gauge the effect of setting these parameters, the SHOW ENGINE INNODB STATUS command reports buffer pool statistics. For details, see Monitoring the Buffer Pool Using the InnoDB

Standard Monitor.

Because the effects of these parameters can vary widely based on your hardware configuration, your data, and the details of your workload, always benchmark to verify the effectiveness before changing these settings in any performance-critical or production environment.

In mixed workloads where most of the activity is OLTP type with periodic batch reporting queries which result in large scans, setting the value of innodb\_old\_blocks\_time during the batch runs can help keep the working set of the normal workload in the buffer pool.

When scanning large tables that cannot fit entirely in the buffer pool, setting innodb\_old\_blocks\_pct to a small value keeps the data that is only read once from consuming a significant portion of the buffer pool. For example, setting innodb\_old\_blocks\_pct=5 restricts this data that is only read once to 5% of the buffer pool.

When scanning small tables that do fit into memory, there is less overhead for moving pages around within the buffer pool, so you can leave innodb\_old\_blocks\_pct at its default value, or even higher, such as innodb\_old\_blocks\_pct=50.

The effect of the innodb\_old\_blocks\_time parameter is harder to predict than the innodb\_old\_blocks\_pct parameter, is relatively small, and varies more with the workload. To arrive at an optimal value, conduct your own benchmarks if the performance improvement from adjusting innodb\_old\_blocks\_pct is not sufficient.

**15.8.3.4** **Configuring** **InnoDB** **Buffer** **Pool** **Prefetching** **(Read-Ahead)**

A read-ahead request is an I/O request to prefetch multiple pages in the buffer pool asynchronously, in anticipation of impending need for these pages. The requests bring in all the pages in one extent. InnoDB uses two read-ahead algorithms to improve I/O performance:

**Linear** read-ahead is a technique that predicts what pages might be needed soon based on pages in the buffer pool being accessed sequentially. You control when InnoDB performs a read-ahead operation by adjusting the number of sequential page accesses required to trigger an asynchronous read request, using the configuration parameter innodb\_read\_ahead\_threshold. Before this parameter was added, InnoDB would only calculate whether to issue an asynchronous prefetch request for the entire next extent when it read the last page of the current extent.

The configuration parameter innodb\_read\_ahead\_threshold controls how sensitive InnoDB is in detecting patterns of sequential page access. If the number of pages read sequentially from an extent is greater than or equal to innodb\_read\_ahead\_threshold, InnoDB initiates an asynchronous read-ahead operation of the entire following extent. innodb\_read\_ahead\_threshold can be set to any value from 0-64. The default value is 56. The higher the value, the more strict the access pattern check. For example, if you set the value to 48, InnoDB triggers a linear read-ahead request only when 48 pages in the current extent have been accessed sequentially. If the value is 8, InnoDB triggers an asynchronous read-ahead even if as few as 8 pages in the extent are accessed sequentially. You can set the value of this parameter in the MySQL configuration file, or change it dynamically with the SET GLOBAL statement, which requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

**Random** read-ahead is a technique that predicts when pages might be needed soon based on pages already in the buffer pool, regardless of the order in which those pages were read. If 13 consecutive pages from the same extent are found in the buffer pool, InnoDB asynchronously issues a request to prefetch the remaining pages of the extent. To enable this feature, set the configuration variable innodb\_random\_read\_ahead to ON.

The SHOW ENGINE INNODB STATUS command displays statistics to help you evaluate the effectiveness of the read-ahead algorithm. Statistics include counter information for the following global status variables:

• Innodb\_buffer\_pool\_read\_ahead

• Innodb\_buffer\_pool\_read\_ahead\_evicted

• Innodb\_buffer\_pool\_read\_ahead\_rnd

This information can be useful when fine-tuning the innodb\_random\_read\_ahead setting.

For more information about I/O performance, see Section 8.5.8, “Optimizing InnoDB Disk I/O” and Section 8.12.1, “Optimizing Disk I/O” .

**15.8.3.5** **Configuring** **Buffer** **Pool** **Flushing**

InnoDB performs certain tasks in the background, including flushing of dirty pages from the buffer pool. Dirty pages are those that have been modified but are not yet written to the data files on disk.

In MySQL 8.0, buffer pool flushing is performed by page cleaner threads. The number of page cleaner threads is controlled by the innodb\_page\_cleaners variable, which has a default value of 4. However, if the number of page cleaner threads exceeds the number of buffer pool instances, innodb\_page\_cleaners is automatically set to the same value as innodb\_buffer\_pool\_instances.

Buffer pool flushing is initiated when the percentage of dirty pages reaches the low water mark value defined by the innodb\_max\_dirty\_pages\_pct\_lwm variable. The default low water mark is 10% of buffer pool pages. A innodb\_max\_dirty\_pages\_pct\_lwm value of 0 disables this early flushing behaviour.

The purpose of the innodb\_max\_dirty\_pages\_pct\_lwm threshold is to control the percentage dirty pages in the buffer pool and to prevent the amount of dirty pages from reaching the threshold defined by the innodb\_max\_dirty\_pages\_pct variable, which has a default value of 90. InnoDB aggressively flushes buffer pool pages if the percentage of dirty pages in the buffer pool reaches the innodb\_max\_dirty\_pages\_pct threshold.

When configuring innodb\_max\_dirty\_pages\_pct\_lwm, the value should always be lower than the innodb\_max\_dirty\_pages\_pct value.

Additional variables permit fine-tuning of buffer pool flushing behavior:

• The innodb\_flush\_neighbors variable defines whether flushing a page from the buffer pool also flushes other dirty pages in the same extent.

• The default setting of 0 disables innodb\_flush\_neighbors. Dirty pages in the same extent are not flushed. This setting is recommended for non-rotational storage (SSD) devices where seek time is not a significant factor.

• A setting of 1 flushes contiguous dirty pages in the same extent.

• A setting of 2 flushes dirty pages in the same extent.

When table data is stored on a traditional HDD storage device, flushing neighbor pages in one operation reduces I/O overhead (primarily for disk seek operations) compared to flushing individual pages at different times. For table data stored on SSD, seek time is not a significant factor and you can disable this setting to spread out write operations.

• The innodb\_lru\_scan\_depth variable specifies, per buffer pool instance, how far down the buffer pool LRU list the page cleaner thread scans looking for dirty pages to flush. This is a background operation performed by a page cleaner thread once per second.

A setting smaller than the default is generally suitable for most workloads. A value that is significantly higher than necessary may impact performance. Only consider increasing the value if you have spare I/O capacity under a typical workload. Conversely, if a write-intensive workload saturates your I/O capacity, decrease the value, especially in the case of a large buffer pool.

When tuning innodb\_lru\_scan\_depth, start with a low value and configure the setting upward with the goal of rarely seeing zero free pages. Also, consider adjusting innodb\_lru\_scan\_depth when changing the number of buffer pool instances, since innodb\_lru\_scan\_depth \* innodb\_buffer\_pool\_instances defines the amount of work performed by the page cleaner thread each second.

The innodb\_flush\_neighbors and innodb\_lru\_scan\_depth variables are primarily intended for write-intensive workloads. With heavy DML activity, flushing can fall behind if it is not aggressive enough, or disk writes can saturate I/O capacity if flushing is too aggressive. The ideal settings depend on your workload, data access patterns, and storage configuration (for example, whether data is stored on HDD or SSD devices).

**Adaptive** **Flushing**

InnoDB uses an adaptive flushing algorithm to dynamically adjust the rate of flushing based on the speed of redo log generation and the current rate of flushing. The intent is to smooth overall performance by ensuring that flushing activity keeps pace with the current workload. Automatically adjusting the flushing rate helps avoid sudden dips in throughput that can occur when bursts of I/O activity due to buffer pool flushing affects the I/O capacity available for ordinary read and write activity.

Sharp checkpoints, which are typically associated with write-intensive workloads that generate a lot of redo entries, can cause a sudden change in throughput, for example. A sharp checkpoint occurs when InnoDB wants to reuse a portion of a log file. Before doing so, all dirty pages with redo entries in that portion of the log file must be flushed. If log files become full, a sharp checkpoint occurs, causing a temporary reduction in throughput. This scenario can occur even if innodb\_max\_dirty\_pages\_pct threshold is not reached.

The adaptive flushing algorithm helps avoid such scenarios by tracking the number of dirty pages in the buffer pool and the rate at which redo log records are being generated. Based on this information, it decides how many dirty pages to flush from the buffer pool each second, which permits it to manage sudden changes in workload.

The innodb\_adaptive\_flushing\_lwm variable defines a low water mark for redo log capacity. When that threshold is crossed, adaptive flushing is enabled, even if the innodb\_adaptive\_flushing variable is disabled.

Internal benchmarking has shown that the algorithm not only maintains throughput over time, but can also improve overall throughput significantly. However, adaptive flushing can affect the I/O pattern of a workload significantly and may not be appropriate in all cases. It gives the most benefit when the redo log is in danger of filling up. If adaptive flushing is not appropriate to the characteristics of your workload, you can disable it. Adaptive flushing controlled by the innodb\_adaptive\_flushing variable, which is enabled by default.

innodb\_flushing\_avg\_loops defines the number of iterations that InnoDB keeps the previously calculated snapshot of the flushing state, controlling how quickly adaptive flushing responds to foreground workload changes. A high innodb\_flushing\_avg\_loops value means that InnoDB keeps the previously calculated snapshot longer, so adaptive flushing responds more slowly.

When setting a high value it is important to ensure that redo log utilization does not reach 75% (the hardcoded limit at which asynchronous flushing starts), and that the innodb\_max\_dirty\_pages\_pct threshold keeps the number of dirty pages to a level that is appropriate for the workload.

Systems with consistent workloads, a large log file size (innodb\_log\_file\_size), and small spikes that do not reach 75% log space utilization should use a high innodb\_flushing\_avg\_loops value to keep flushing as smooth as possible. For systems with extreme load spikes or log files that do not provide a lot of space, a smaller value allows flushing to closely track workload changes, and helps to avoid reaching 75% log space utilization.

Be aware that if flushing falls behind, the rate of buffer pool flushing can exceed the I/O capacity available to InnoDB, as defined by innodb\_io\_capacity setting. The innodb\_io\_capacity\_max

value defines an upper limit on I/O capacity in such situations, so that a spike in I/O activity does not consume the entire I/O capacity of the server.

The innodb\_io\_capacity setting is applicable to all buffer pool instances. When dirty pages are flushed, I/O capacity is divided equally among buffer pool instances.

**Limiting** **Buffer** **Flushing** **During** **Idle** **Periods**

As of MySQL 8.0. 18, you can use the innodb\_idle\_flush\_pct variable to limit the rate of

buffer pool flushing during idle periods, which are periods of time that database pages are not modified. The innodb\_idle\_flush\_pct value is a percentage of the innodb\_io\_capacity setting, which defines the number of I/O operations per second available to InnoDB. The default innodb\_idle\_flush\_pct value is 100, which is 100 percent of the innodb\_io\_capacity setting. To limit flushing during idle periods, define an innodb\_idle\_flush\_pct value less than 100.

Limiting page flushing during idle periods can help extend the life of solid state storage devices. Side effects of limiting page flushing during idle periods may include a longer shutdown time following a lengthy idle period, and a longer recovery period should a server failure occur.

**15.8.3.6** **Saving** **and** **Restoring** **the** **Buffer** **Pool** **State**

To reduce the warmup period after restarting the server, InnoDB saves a percentage of the most recently used pages for each buffer pool at server shutdown and restores these pages at server startup. The percentage of recently used pages that is stored is defined by the innodb\_buffer\_pool\_dump\_pct configuration option.

After restarting a busy server, there is typically a warmup period with steadily increasing throughput, as disk pages that were in the buffer pool are brought back into memory (as the same data is queried, updated, and so on). The ability to restore the buffer pool at startup shortens the warmup period by reloading disk pages that were in the buffer pool before the restart rather than waiting for DML operations to access corresponding rows. Also, I/O requests can be performed in large batches, making the overall I/O faster. Page loading happens in the background, and does not delay database startup.

In addition to saving the buffer pool state at shutdown and restoring it at startup, you can save and restore the buffer pool state at any time, while the server is running. For example, you can save the state of the buffer pool after reaching a stable throughput under a steady workload. You could also restore the previous buffer pool state after running reports or maintenance jobs that bring data pages into the buffer pool that are only requited for those operations, or after running some other non-typical workload.

Even though a buffer pool can be many gigabytes in size, the buffer pool data that InnoDB saves to disk is tiny by comparison. Only tablespace IDs and page IDs necessary to locate the appropriate pages are saved to disk. This information is derived from the INNODB\_BUFFER\_PAGE\_LRU INFORMATION\_SCHEMA table. By default, tablespace ID and page ID data is saved in a file named ib\_buffer\_pool, which is saved to the InnoDB data directory. The file name and location can be modified using the innodb\_buffer\_pool\_filename configuration parameter.

Because data is cached in and aged out of the buffer pool as it is with regular database operations, there is no problem if the disk pages are recently updated, or if a DML operation involves data that has not yet been loaded. The loading mechanism skips requested pages that no longer exist.

The underlying mechanism involves a background thread that is dispatched to perform the dump and load operations.

Disk pages from compressed tables are loaded into the buffer pool in their compressed form. Pages are uncompressed as usual when page contents are accessed during DML operations. Because uncompressing pages is a CPU-intensive process, it is more efficient for concurrency to perform the operation in a connection thread rather than in the single thread that performs the buffer pool restore operation.

Operations related to saving and restoring the buffer pool state are described in the following topics:

• [Configuring the Dump Percentage for Buffer Pool Pages](#_bookmark23)

• [Saving the Buffer Pool State at Shutdown and Restoring it at Startup](#_bookmark24)

• [Saving and Restoring the Buffer Pool State Online](#_bookmark25)

• [Displaying Buffer Pool Dump Progress](#_bookmark26)

• [Displaying Buffer Pool Load Progress](#_bookmark27)

• [Aborting a Buffer Pool Load Operation](#_bookmark28)

• [Monitoring Buffer Pool Load Progress Using Performance Schema](#_bookmark29)

**Configuring** **the** **Dump** **Percentage** **for** **Buffer** **Pool** **Pages**

Before dumping pages from the buffer pool, you can configure the percentage of most-recently- used buffer pool pages that you want to dump by setting the innodb\_buffer\_pool\_dump\_pct option. If you plan to dump buffer pool pages while the server is running, you can configure the option dynamically:

SET GLOBAL innodb\_buffer\_pool\_dump\_pct=40;

If you plan to dump buffer pool pages at server shutdown, set innodb\_buffer\_pool\_dump\_pct in your configuration file.

[mysqld]

innodb\_buffer\_pool\_dump\_pct=40

The innodb\_buffer\_pool\_dump\_pct default value is 25 (dump 25% of most-recently-used pages).

**Saving** **the** **Buffer** **Pool** **State** **at** **Shutdown** **and** **Restoring** **it** **at** **Startup**

To save the state of the buffer pool at server shutdown, issue the following statement prior to shutting down the server:

SET GLOBAL innodb\_buffer\_pool\_dump\_at\_shutdown=ON;

innodb\_buffer\_pool\_dump\_at\_shutdown is enabled by default.

To restore the buffer pool state at server startup, specify the --innodb-buffer-pool-load-at- startup option when starting the server:

mysqld --innodb-buffer-pool-load-at-startup=ON;

innodb\_buffer\_pool\_load\_at\_startup is enabled by default.

**Saving** **and** **Restoring** **the** **Buffer** **Pool** **State** **Online**

To save the state of the buffer pool while MySQL server is running, issue the following statement: SET GLOBAL innodb\_buffer\_pool\_dump\_now=ON; To restore the buffer pool state while MySQL is running, issue the following statement: SET GLOBAL innodb\_buffer\_pool\_load\_now=ON;

**Displaying** **Buffer** **Pool** **Dump** **Progress**

To display progress when saving the buffer pool state to disk, issue the following statement: SHOW STATUS LIKE 'Innodb\_buffer\_pool\_dump\_status';

**WHERE** **NAME** **LIKE** **'stage/innodb/buffer%';**

**WHERE** **NAME** **LIKE** **'%stages%';**

If the operation has not yet started, “not started” is returned. If the operation is complete, the completion time is printed (e.g. Finished at 110505 12:18:02). If the operation is in progress, status information is provided (e.g. Dumping buffer pool 5/7, page 237/2873).

**Displaying** **Buffer** **Pool** **Load** **Progress**

To display progress when loading the buffer pool, issue the following statement: SHOW STATUS LIKE 'Innodb\_buffer\_pool\_load\_status';

If the operation has not yet started, “not started” is returned. If the operation is complete, the completion time is printed (e.g. Finished at 110505 12:23:24). If the operation is in progress, status information is provided (e.g. Loaded 123/22301 pages).

**Aborting** **a** **Buffer** **Pool** **Load** **Operation**

To abort a buffer pool load operation, issue the following statement: SET GLOBAL innodb\_buffer\_pool\_load\_abort=ON;

**Monitoring** **Buffer** **Pool** **Load** **Progress** **Using** **Performance** **Schema**

You can monitor buffer pool load progress using Performance Schema.

The following example demonstrates how to enable the stage/innodb/buffer pool load stage event instrument and related consumer tables to monitor buffer pool load progress.

For information about buffer pool dump and load procedures used in this example, see [Section 15.8.3.6, “Saving and Restoring the Buffer Pool State”](#_bookmark22) . For information about Performance Schema stage event instruments and related consumers, see Section 27.12.5, “Performance Schema Stage Event Tables” .

1. Enable the stage/innodb/buffer pool load instrument:

mysql> **UPDATE** **performance\_schema.setup\_instruments** **SET** **ENABLED** **=** **'YES'**

2. Enable the stage event consumer tables, which include events\_stages\_current, events\_stages\_history, and events\_stages\_history\_long.

mysql> **UPDATE** **performance\_schema.setup\_consumers** **SET** **ENABLED** **=** **'YES'**

3. Dump the current buffer pool state by enabling innodb\_buffer\_pool\_dump\_now. mysql> **SET** **GLOBAL** **innodb\_buffer\_pool\_dump\_now=ON;**

4. Check the buffer pool dump status to ensure that the operation has completed.

mysql> **SHOW** **STATUS** **LIKE** **'Innodb\_buffer\_pool\_dump\_status'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Variable\_name: Innodb\_buffer\_pool\_dump\_status

Value: Buffer pool(s) dump completed at 150202 16:38:58

5. Load the buffer pool by enabling innodb\_buffer\_pool\_load\_now:

mysql> **SET** **GLOBAL** **innodb\_buffer\_pool\_load\_now=ON;**

6. Check the current status of the buffer pool load operation by querying the Performance Schema events\_stages\_current table. The WORK\_COMPLETED column shows the number of buffer pool pages loaded. The WORK\_ESTIMATED column provides an estimate of the remaining work, in pages.

mysql> **SELECT** **EVENT\_NAME,** **WORK\_COMPLETED,** **WORK\_ESTIMATED**

**FROM** **performance\_schema** **.events\_stages\_current;**

+-------------------------------+----------------+----------------+



| EVENT\_NAME | WORK\_COMPLETED | WORK\_ESTIMATED |

+-------------------------------+----------------+----------------+

| stage/innodb/buffer pool load | 5353 | 7167 |

+-------------------------------+----------------+----------------+

The events\_stages\_current table returns an empty set if the buffer pool load operation has completed. In this case, you can check the events\_stages\_history table to view data for the completed event. For example:

mysql> **SELECT** **EVENT\_NAME,** **WORK\_COMPLETED,** **WORK\_ESTIMATED**

**FROM** **performance\_schema** **.events\_stages\_history;**

+-------------------------------+----------------+----------------+

| EVENT\_NAME | WORK\_COMPLETED | WORK\_ESTIMATED |

+-------------------------------+----------------+----------------+

| stage/innodb/buffer pool load | 7167 | 7167 |

+-------------------------------+----------------+----------------+

**Note**

You can also monitor buffer pool load progress using Performance Schema when loading the buffer pool at startup using innodb\_buffer\_pool\_load\_at\_startup. In this case, the stage/ innodb/buffer pool load instrument and related consumers must be enabled at startup. For more information, see Section 27.3, “Performance Schema Startup Configuration” .

**15.8.3.7** **Excluding** **Buffer** **Pool** **Pages** **from** **Core** **Files**

A core file records the status and memory image of a running process. Because the buffer pool resides in main memory, and the memory image of a running process is dumped to the core file, systems with large buffer pools can produce large core files when the mysqld process dies.

Large core files can be problematic for a number of reasons including the time it takes to write them, the amount of disk space they consume, and the challenges associated with transferring large files.

To reduce core file size, you can disable the innodb\_buffer\_pool\_in\_core\_file variable to omit buffer pool pages from core dumps. The innodb\_buffer\_pool\_in\_core\_file variable was introduced in MySQL 8.0.14 and is enabled by default.

Excluding buffer pool pages may also be desirable from a security perspective if you have concerns about dumping database pages to core files that may be shared inside or outside of your organization for debugging purposes.

**Note**

Access to the data present in buffer pool pages at the time the mysqld process died may be beneficial in some debugging scenarios. If in doubt whether to include or exclude buffer pool pages, consult MySQL Support.

Disabling innodb\_buffer\_pool\_in\_core\_file takes effect only if the core\_file variable is enabled and the operating system supports the MADV\_DONTDUMP non-POSIX extension to the [madvise()](http://man7.org/linux/man-pages/man2/madvise.2.html) system call, which is supported in Linux 3.4 and later. The MADV\_DONTDUMP extension causes pages in a specified range to be excluded from core dumps.

Assuming the operating system supports the MADV\_DONTDUMP extension, start the server with the --core-file and --innodb-buffer-pool-in-core-file=OFF options to generate core files without buffer pool pages.

$> mysqld --core-file --innodb-buffer-pool-in-core-file=OFF

The core\_file variable is read only and disabled by default. It is enabled by specifying the --core- file option at startup. The innodb\_buffer\_pool\_in\_core\_file variable is dynamic. It can be specified at startup or configured at runtime using a SET statement.

mysql> SET GLOBAL innodb\_buffer\_pool\_in\_core\_file=OFF;

If the innodb\_buffer\_pool\_in\_core\_file variable is disabled but MADV\_DONTDUMP is not supported by the operating system, or an madvise() failure occurs, a warning is written to the MySQL server error log and the core\_file variable is disabled to prevent writing core files that unintentionally include buffer pool pages. If the read-only core\_file variable becomes disabled, the server must be restarted to enable it again.

The following table shows configuration and MADV\_DONTDUMP support scenarios that determine whether core files are generated and whether they include buffer pool pages.

**Table** **15.4** **Core** **File** **Configuration** **Scenarios**

|  |  |  |  |
| --- | --- | --- | --- |
| **core\_file** **variable** | **innodb\_buffer\_pool\_**  **variable** | **\_****e(\_)file** **MADV\_DONTDUMP** **Support** | **Outcome** |
| OFF (default) | Not relevant to outcome | Not relevant to outcome | Core file is not  generated |
| ON | ON (default) | Not relevant to outcome | Core file is generated with buffer pool pages |
| ON | OFF | Yes | Core file is generated without buffer pool pages |
| ON | OFF | No | Core file is not generated, core\_file is disabled, and a warning is written to the server error log |

The reduction in core file size achieved by disabling the innodb\_buffer\_pool\_in\_core\_file variable depends on the size of the buffer pool, but it is also affected by the InnoDB page size. A smaller page size means more pages are required for the same amount of data, and more pages means more page metadata. The following table provides size reduction examples that you might see for a 1GB buffer pool with different pages sizes.

**Table** **15.5** **Core** **File** **Size** **with** **Buffer** **Pool** **Pages** **Included** **and** **Excluded**

|  |  |  |
| --- | --- | --- |
| **innodb\_page\_size** **Setting** | **Buffer** **Pool** **Pages** **Included** **(innodb\_buffer\_pool\_in\_core**  **Buffer** **Pool** **Pages** **Excluded** **(innodb\_buffer\_pool\_in\_core\_fil** | **\_file=ON)** |
| 4KB | 2.1GB | 0.9GB |
| 64KB | 1.7GB | 0.7GB |

**15.8.4** **Configuring** **Thread** **Concurrency** **for** **InnoDB**

InnoDB uses operating system threads to process requests from user transactions. (Transactions may issue many requests to InnoDB before they commit or roll back.) On modern operating systems and servers with multi-core processors, where context switching is efficient, most workloads run well without any limit on the number of concurrent threads.

In situations where it is helpful to minimize context switching between threads, InnoDB can use a number of techniques to limit the number of concurrently executing operating system threads (and thus the number of requests that are processed at any one time). When InnoDB receives a new request from a user session, if the number of threads concurrently executing is at a pre-defined limit, the new request sleeps for a short time before it tries again. A request that cannot be rescheduled after the sleep is put in a first-in/first-out queue and eventually is processed. Threads waiting for locks are not counted in the number of concurrently executing threads.

You can limit the number of concurrent threads by setting the configuration parameter innodb\_thread\_concurrency. Once the number of executing threads reaches this limit, additional threads sleep for a number of microseconds, set by the configuration parameter innodb\_thread\_sleep\_delay, before being placed into the queue.

You can set the configuration option innodb\_adaptive\_max\_sleep\_delay to the highest

value you would allow for innodb\_thread\_sleep\_delay, and InnoDB automatically adjusts innodb\_thread\_sleep\_delay up or down depending on the current thread-scheduling activity. This dynamic adjustment helps the thread scheduling mechanism to work smoothly during times when the system is lightly loaded and when it is operating near full capacity.

The default value for innodb\_thread\_concurrency and the implied default limit on the number of concurrent threads has been changed in various releases of MySQL and InnoDB. The default value of innodb\_thread\_concurrency is 0, so that by default there is no limit on the number of concurrently executing threads.

InnoDB causes threads to sleep only when the number of concurrent threads is limited. When there is no limit on the number of threads, all contend equally to be scheduled. That is, if innodb\_thread\_concurrency is 0, the value of innodb\_thread\_sleep\_delay is ignored.

When there is a limit on the number of threads (when innodb\_thread\_concurrency is >

0), InnoDB reduces context switching overhead by permitting multiple requests made during the execution of a *single* *SQL* *statement* to enter InnoDB without observing the limit set by innodb\_thread\_concurrency. Since an SQL statement (such as a join) may comprise multiple row operations within InnoDB, InnoDB assigns a specified number of “tickets” that allow a thread to be scheduled repeatedly with minimal overhead.

When a new SQL statement starts, a thread has no tickets, and it must observe innodb\_thread\_concurrency. Once the thread is entitled to enter InnoDB, it is assigned a number of tickets that it can use for subsequently entering InnoDB to perform row operations. If the tickets run out, the thread is evicted, and innodb\_thread\_concurrency is observed again which may place the thread back into the first-in/first-out queue of waiting threads. When the thread is once again entitled to enter InnoDB, tickets are assigned again. The number of tickets assigned is specified by the global option innodb\_concurrency\_tickets, which is 5000 by default. A thread that is waiting for a lock is given one ticket once the lock becomes available.

The correct values of these variables depend on your environment and workload. Try a range of different values to determine what value works for your applications. Before limiting the number of concurrently executing threads, review configuration options that may improve the performance of InnoDB on multi-core and multi-processor computers, such as innodb\_adaptive\_hash\_index.

For general performance information about MySQL thread handling, see Section 5.1.12.1, “Connection

Interfaces” .

**15.8.5** **Configuring** **the** **Number** **of** **Background** **InnoDB** **I/O** **Threads**

InnoDB uses background threads to service various types of I/O requests. You can configure the number of background threads that service read and write I/O on data pages using the innodb\_read\_io\_threads and innodb\_write\_io\_threads configuration parameters. These parameters signify the number of background threads used for read and write requests, respectively. They are effective on all supported platforms. You can set values for these parameters in the MySQL option file (my.cnf or my.ini); you cannot change values dynamically. The default value for these parameters is 4 and permissible values range from 1-64.

The purpose of these configuration options to make InnoDB more scalable on high end systems. Each background thread can handle up to 256 pending I/O requests. A major source of background I/O is read-ahead requests. InnoDB tries to balance the load of incoming requests in such way that most background threads share work equally. InnoDB also attempts to allocate read requests from the same extent to the same thread, to increase the chances of coalescing the requests. If you have a high

end I/O subsystem and you see more than 64 × innodb\_read\_io\_threads pending read requests in SHOW ENGINE INNODB STATUS output, you might improve performance by increasing the value of innodb\_read\_io\_threads.

On Linux systems, InnoDB uses the asynchronous I/O subsystem by default to perform read-ahead and write requests for data file pages, which changes the way that InnoDB background threads service these types of I/O requests. For more information, see [Section 15.8.6, “Using Asynchronous I/O on](#_bookmark31) [Linux”](#_bookmark31) .

For more information about InnoDB I/O performance, see Section 8.5.8, “Optimizing InnoDB Disk I/O” .

**15.8.6** **Using** **Asynchronous** **I/O** **on** **Linux**

InnoDB uses the asynchronous I/O subsystem (native AIO) on Linux to perform read-ahead and write requests for data file pages. This behavior is controlled by the innodb\_use\_native\_aio configuration option, which applies to Linux systems only and is enabled by default. On other Unix- like systems, InnoDB uses synchronous I/O only. Historically, InnoDB only used asynchronous I/O on Windows systems. Using the asynchronous I/O subsystem on Linux requires the libaio library.

With synchronous I/O, query threads queue I/O requests, and InnoDB background threads retrieve the queued requests one at a time, issuing a synchronous I/O call for each. When an I/O request is completed and the I/O call returns, the InnoDB background thread that is handling the request calls an I/O completion routine and returns to process the next request. The number of requests that can be processed in parallel is *n*, where *n* is the number of InnoDB background threads.

The number of InnoDB background threads is controlled by innodb\_read\_io\_threads and innodb\_write\_io\_threads. See [Section 15.8.5, “Configuring the Number of Background InnoDB I/](#_bookmark30) [O Threads”](#_bookmark30) .

With native AIO, query threads dispatch I/O requests directly to the operating system, thereby removing the limit imposed by the number of background threads. InnoDB background threads wait for I/O events to signal completed requests. When a request is completed, a background thread calls an I/ O completion routine and resumes waiting for I/O events.

The advantage of native AIO is scalability for heavily I/O-bound systems that typically show many pending reads/writes in SHOW ENGINE INNODB STATUS\G output. The increase in parallel processing when using native AIO means that the type of I/O scheduler or properties of the disk array controller have a greater influence on I/O performance.

A potential disadvantage of native AIO for heavily I/O-bound systems is lack of control over the number of I/O write requests dispatched to the operating system at once. Too many I/O write requests dispatched to the operating system for parallel processing could, in some cases, result in I/O read starvation, depending on the amount of I/O activity and system capabilities.

If a problem with the asynchronous I/O subsystem in the OS prevents InnoDB from starting, you can start the server with innodb\_use\_native\_aio=0. This option may also be disabled automatically during startup if InnoDB detects a potential problem such as a combination of tmpdir location, tmpfs file system, and Linux kernel that does not support asynchronous I/O on tmpfs.

**15.8.7** **Configuring** **InnoDB** **I/O** **Capacity**

The InnoDB master thread and other threads perform various tasks in the background, most of which are I/O related, such as flushing dirty pages from the buffer pool and writing changes from the change buffer to the appropriate secondary indexes. InnoDB attempts to perform these tasks in a way that does not adversely affect the normal working of the server. It tries to estimate the available I/O bandwidth and tune its activities to take advantage of available capacity.

The innodb\_io\_capacity variable defines the overall I/O capacity available to InnoDB. It should be set to approximately the number of I/O operations that the system can perform per second (IOPS). When innodb\_io\_capacity is set, InnoDB estimates the I/O bandwidth available for background tasks based on the set value.

You can set innodb\_io\_capacity to a value of 100 or greater. The default value is 200. Typically, values around 100 are appropriate for consumer-level storage devices, such as hard drives up to 7200 RPMs. Faster hard drives, RAID configurations, and solid state drives (SSDs) benefit from higher values.

Ideally, keep the setting as low as practical, but not so low that background activities fall behind. If the value is too high, data is removed from the buffer pool and change buffer too quickly for caching to provide a significant benefit. For busy systems capable of higher I/O rates, you can set a higher value to help the server handle the background maintenance work associated with a high rate of row changes. Generally, you can increase the value as a function of the number of drives used for InnoDB I/O. For example, you can increase the value on systems that use multiple disks or SSDs.

The default setting of 200 is generally sufficient for a lower-end SSD. For a higher-end, bus-attached SSD, consider a higher setting such as 1000, for example. For systems with individual 5400 RPM or

7200 RPM drives, you might lower the value to 100, which represents an estimated proportion of the I/ O operations per second (IOPS) available to older-generation disk drives that can perform about 100 IOPS.

Although you can specify a high value such as a million, in practice such large values have little benefit. Generally, a value higher than 20000 is not recommended unless you are certain that lower values are insufficient for your workload.

Consider write workload when tuning innodb\_io\_capacity. Systems with large write workloads are likely to benefit from a higher setting. A lower setting may be sufficient for systems with a small write workload.

The innodb\_io\_capacity setting is not a per buffer pool instance setting. Available I/O capacity is distributed equally among buffer pool instances for flushing activities.

You can set the innodb\_io\_capacity value in the MySQL option file (my.cnf or my.ini) or modify it at runtime using a SET GLOBAL statement, which requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

**Ignoring** **I/O** **Capacity** **at** **Checkpoints**

The innodb\_flush\_sync variable, which is enabled by default, causes the innodb\_io\_capacity setting to be ignored during bursts of I/O activity that occur at checkpoints. To adhere to the I/O rate defined by the innodb\_io\_capacity setting, disable innodb\_flush\_sync.

You can set the innodb\_flush\_sync value in the MySQL option file (my.cnf or my.ini) or modify it at runtime using a SET GLOBAL statement, which requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

**Configuring** **an** **I/O** **Capacity** **Maximum**

If flushing activity falls behind, InnoDB can flush more aggressively, at a higher rate of I/ O operations per second (IOPS) than defined by the innodb\_io\_capacity variable. The innodb\_io\_capacity\_max variable defines a maximum number of IOPS performed by InnoDB background tasks in such situations.

If you specify an innodb\_io\_capacity setting at startup but do not specify a value for innodb\_io\_capacity\_max, innodb\_io\_capacity\_max defaults to twice the value of innodb\_io\_capacity or 2000, whichever value is greater.

When configuring innodb\_io\_capacity\_max, twice the innodb\_io\_capacity is often a good starting point. The default value of 2000 is intended for workloads that use an SSD or more than one regular disk drive. A setting of 2000 is likely too high for workloads that do not use SSDs or multiple disk drives, and could allow too much flushing. For a single regular disk drive, a setting between 200 and 400 is recommended. For a high-end, bus-attached SSD, consider a higher setting such as 2500. As with the innodb\_io\_capacity setting, keep the setting as low as practical, but not so low that InnoDB cannot sufficiently extend rate of IOPS beyond the innodb\_io\_capacity setting.

Consider write workload when tuning innodb\_io\_capacity\_max. Systems with large write workloads may benefit from a higher setting. A lower setting may be sufficient for systems with a small write workload.

innodb\_io\_capacity\_max cannot be set to a value lower than the innodb\_io\_capacity value.

Setting innodb\_io\_capacity\_max to DEFAULT using a SET statement (SET GLOBAL innodb\_io\_capacity\_max=DEFAULT) sets innodb\_io\_capacity\_max to the maximum value.

The innodb\_io\_capacity\_max limit applies to all buffer pool instances. It is not a per buffer pool instance setting.

**15.8.8** **Configuring** **Spin** **Lock** **Polling**

InnoDB mutexes and rw-locks are typically reserved for short intervals. On a multi-core system, it can be more efficient for a thread to continuously check if it can acquire a mutex or rw-lock for a period of time before it sleeps. If the mutex or rw-lock becomes available during this period, the thread can continue immediately, in the same time slice. However, too-frequent polling of a shared object such as a mutex or rw-lock by multiple threads can cause “cache ping pong” , which results in processors invalidating portions of each other's cache. InnoDB minimizes this issue by forcing a random delay between polls to desynchronize polling activity. The random delay is implemented as a spin-wait loop.

The duration of a spin-wait loop is determined by the number of PAUSE instructions that occur in the loop. That number is generated by randomly selecting an integer ranging from 0 up to but not including the innodb\_spin\_wait\_delay value, and multiplying that value by 50. (The multiplier value, 50, is hardcoded before MySQL 8.0.16, and configurable thereafter.) For example, an integer is randomly selected from the following range for an innodb\_spin\_wait\_delay setting of 6:

{0,1,2,3,4,5}

The selected integer is multiplied by 50, resulting in one of six possible PAUSE instruction values:

{0,50,100,150,200,250}

For that set of values, 250 is the maximum number of PAUSE instructions that can occur in a spin- wait loop. An innodb\_spin\_wait\_delay setting of 5 results in a set of five possible values {0,50,100,150,200}, where 200 is the maximum number of PAUSE instructions, and so on. In this way, the innodb\_spin\_wait\_delay setting controls the maximum delay between spin lock polls.

On a system where all processor cores share a fast cache memory, you might reduce the maximum delay or disable the busy loop altogether by setting innodb\_spin\_wait\_delay=0. On a system with multiple processor chips, the effect of cache invalidation can be more significant and you might increase the maximum delay.

In the 100MHz Pentium era, an innodb\_spin\_wait\_delay unit was calibrated to be equivalent to one microsecond. That time equivalence did not hold, but PAUSE instruction duration remained fairly constant in terms of processor cycles relative to other CPU instructions until the introduction of the Skylake generation of processors, which have a comparatively longer PAUSE instruction. The innodb\_spin\_wait\_pause\_multiplier variable was introduced in MySQL 8.0.16 to provide a way to account for differences in PAUSE instruction duration.

The innodb\_spin\_wait\_pause\_multiplier variable controls the size of PAUSE instruction values. For example, assuming an innodb\_spin\_wait\_delay setting of 6, decreasing the innodb\_spin\_wait\_pause\_multiplier value from 50 (the default and previously hardcoded value) to 5 generates a set of smaller PAUSE instruction values:

{0,5,10,15,20,25}

The ability to increase or decrease PAUSE instruction values permits fine tuning InnoDB for different processor architectures. Smaller PAUSE instruction values would be appropriate for processor architectures with a comparatively longer PAUSE instruction, for example.

The innodb\_spin\_wait\_delay and innodb\_spin\_wait\_pause\_multiplier variables are dynamic. They can be specified in a MySQL option file or modified at runtime using a SET GLOBAL statement. Modifying the variables at runtime requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

**15.8.9** **Purge** **Configuration**

InnoDB does not physically remove a row from the database immediately when you delete it with an SQL statement. A row and its index records are only physically removed when InnoDB discards the undo log record written for the deletion. This removal operation, which only occurs after the row is no longer required for multi-version concurrency control (MVCC) or rollback, is called a purge.

Purge runs on a periodic schedule. It parses and processes undo log pages from the history list, which is a list of undo log pages for committed transactions that is maintained by the InnoDB transaction system. Purge frees the undo log pages from the history list after processing them.

**Configuring** **Purge** **Threads**

Purge operations are performed in the background by one or more purge threads. The number of purge threads is controlled by the innodb\_purge\_threads variable. The default value is 4.

If DML action is concentrated on a single table, purge operations for the table are performed by a single purge thread, which can result in slowed purge operations, increased purge lag, and increased tablespace file size if the DML operations involve large object values. From MySQL 8.0.26, if the innodb\_max\_purge\_lag setting is exceeded, purge work is automatically redistributed among available purge threads. Too many active purge threads in this scenario can cause

contention with user threads, so manage the innodb\_purge\_threads setting accordingly. The innodb\_max\_purge\_lag variable is set to 0 by default, which means that there is no maximum purge lag by default.

If DML action is concentrated on few tables, keep the innodb\_purge\_threads setting low so that the threads do not contend with each other for access to the busy tables. If DML operations are spread across many tables, consider a higher innodb\_purge\_threads setting. The maximum number of purge threads is 32.

The innodb\_purge\_threads setting is the maximum number of purge threads permitted. The purge system automatically adjusts the number of purge threads that are used.

**Configuring** **Purge** **Batch** **Size**

The innodb\_purge\_batch\_size variable defines the number of undo log pages that purge parses and processes in one batch from the history list. The default value is 300. In a multithreaded purge configuration, the coordinator purge thread divides innodb\_purge\_batch\_size by innodb\_purge\_threads and assigns that number of pages to each purge thread.

The purge system also frees the undo log pages that are no longer required. It does so every 128 iterations through the undo logs. In addition to defining the number of undo log pages parsed and processed in a batch, the innodb\_purge\_batch\_size variable defines the number of undo log pages that purge frees every 128 iterations through the undo logs.

The innodb\_purge\_batch\_size variable is intended for advanced performance tuning and experimentation. Most users need not change innodb\_purge\_batch\_size from its default value.

**Configuring** **the** **Maximum** **Purge** **Lag**

The innodb\_max\_purge\_lag variable defines the desired maximum purge lag. When the purge lag exceeds the innodb\_max\_purge\_lag threshold, a delay is imposed on INSERT, UPDATE, and DELETE operations to allow time for purge operations to catch up. The default value is 0, which means there is no maximum purge lag and no delay.

The InnoDB transaction system maintains a list of transactions that have index records delete-marked by UPDATE or DELETE operations. The length of the list is the purge lag. Prior to MySQL 8.0.14, the purge lag delay is calculated by the following formula, which results in a minimum delay of 5000 microseconds:

(purge lag/innodb\_max\_purge\_lag - 0.5) \* 10000

As of MySQL 8.0.14, the purge lag delay is calculated by the following revised formula, which reduces the minimum delay to 5 microseconds. A delay of 5 microseconds is more appropriate for modern systems.

(purge\_lag/innodb\_max\_purge\_lag - 0.9995) \* 10000

The delay is calculated at the beginning of a purge batch.

A typical innodb\_max\_purge\_lag setting for a problematic workload might be 1000000 (1 million), assuming that transactions are small, only 100 bytes in size, and it is permissible to have 100MB of unpurged table rows.

The purge lag is presented as the History list length value in the TRANSACTIONS section of SHOW ENGINE INNODB STATUS output.

mysql> SHOW ENGINE INNODB STATUS;

...

------------

TRANSACTIONS

------------

Trx id counter 0 290328385

Purge done for trx's n:o < 0 290315608 undo n:o < 0 17

History list length 20

The History list length is typically a low value, usually less than a few thousand, but a write- heavy workload or long running transactions can cause it to increase, even for transactions that are read only. The reason that a long running transaction can cause the History list length to increase is that under a consistent read transaction isolation level such as REPEATABLE READ, a transaction must return the same result as when the read view for that transaction was created. Consequently, the InnoDB multi-version concurrency control (MVCC) system must keep a copy of the data in the undo log until all transactions that depend on that data have completed. The following are examples of long running transactions that could cause the History list length to increase:

• A mysqldump operation that uses the --single-transaction option while there is a significant amount of concurrent DML.

• Running a SELECT query after disabling autocommit, and forgetting to issue an explicit COMMIT or ROLLBACK.

To prevent excessive delays in extreme situations where the purge lag becomes huge, you can limit the delay by setting the innodb\_max\_purge\_lag\_delay variable. The innodb\_max\_purge\_lag\_delay variable specifies the maximum delay in microseconds for the delay imposed when the innodb\_max\_purge\_lag threshold is exceeded. The specified innodb\_max\_purge\_lag\_delay value is an upper limit on the delay period calculated by the innodb\_max\_purge\_lag formula.

**Purge** **and** **Undo** **Tablespace** **Truncation**

The purge system is also responsible for truncating undo tablespaces. You can configure the innodb\_purge\_rseg\_truncate\_frequency variable to control the frequency with which the purge system looks for undo tablespaces to truncate. For more information, see Truncating Undo Tablespaces.

**15.8.10** **Configuring** **Optimizer** **Statistics** **for** **InnoDB**

This section describes how to configure persistent and non-persistent optimizer statistics for InnoDB tables.

Persistent optimizer statistics are persisted across server restarts, allowing for greater plan stability and more consistent query performance. Persistent optimizer statistics also provide control and flexibility with these additional benefits:

• You can use the innodb\_stats\_auto\_recalc configuration option to control whether statistics are updated automatically after substantial changes to a table.

• You can use the STATS\_PERSISTENT, STATS\_AUTO\_RECALC, and STATS\_SAMPLE\_PAGES clauses with CREATE TABLE and ALTER TABLE statements to configure optimizer statistics for individual tables.

• You can query optimizer statistics data in the mysql.innodb\_table\_stats and mysql.innodb\_index\_stats tables.

• You can view the last\_update column of the mysql.innodb\_table\_stats and mysql.innodb\_index\_stats tables to see when statistics were last updated.

• You can manually modify the mysql.innodb\_table\_stats and mysql.innodb\_index\_stats tables to force a specific query optimization plan or to test alternative plans without modifying the

database.

The persistent optimizer statistics feature is enabled by default (innodb\_stats\_persistent=ON).

Non-persistent optimizer statistics are cleared on each server restart and after some other operations,

and recomputed on the next table access. As a result, different estimates could be produced when recomputing statistics, leading to different choices in execution plans and variations in query performance.

This section also provides information about estimating ANALYZE TABLE complexity, which may be useful when attempting to achieve a balance between accurate statistics and ANALYZE TABLE execution time.

**15.8.10.1** **Configuring** **Persistent** **Optimizer** **Statistics** **Parameters**

The persistent optimizer statistics feature improves plan stability by storing statistics to disk and making them persistent across server restarts so that the optimizer is more likely to make consistent choices each time for a given query.

Optimizer statistics are persisted to disk when innodb\_stats\_persistent=ON or when individual tables are defined with STATS\_PERSISTENT=1. innodb\_stats\_persistent is enabled by default.

Formerly, optimizer statistics were cleared when restarting the server and after some other types of operations, and recomputed on the next table access. Consequently, different estimates could be produced when recalculating statistics leading to different choices in query execution plans and variation in query performance.

Persistent statistics are stored in the mysql.innodb\_table\_stats and mysql.innodb\_index\_stats tables. See [InnoDB Persistent Statistics Tables](#_bookmark32).

If you prefer not to persist optimizer statistics to disk, see Section 15.8.10.2, “Configuring Non- Persistent Optimizer Statistics Parameters”

**Configuring** **Automatic** **Statistics** **Calculation** **for** **Persistent** **Optimizer** **Statistics**

The innodb\_stats\_auto\_recalc variable, which is enabled by default, controls whether statistics are calculated automatically when a table undergoes changes to more than 10% of its rows. You can also configure automatic statistics recalculation for individual tables by specifying the STATS\_AUTO\_RECALC clause when creating or altering a table.

Because of the asynchronous nature of automatic statistics recalculation, which occurs in the background, statistics may not be recalculated instantly after running a DML operation that affects

more than 10% of a table, even when innodb\_stats\_auto\_recalc is enabled. Statistics recalculation can be delayed by few seconds in some cases. If up-to-date statistics are required immediately, run ANALYZE TABLE to initiate a synchronous (foreground) recalculation of statistics.

If innodb\_stats\_auto\_recalc is disabled, you can ensure the accuracy of optimizer statistics by executing the ANALYZE TABLE statement after making substantial changes to indexed columns. You might also consider adding ANALYZE TABLE to setup scripts that you run after loading data, and running ANALYZE TABLE on a schedule at times of low activity.

When an index is added to an existing table, or when a column is added or dropped, index statistics are calculated and added to the innodb\_index\_stats table regardless of the value of innodb\_stats\_auto\_recalc.

**Configuring** **Optimizer** **Statistics** **Parameters** **for** **Individual** **Tables**

innodb\_stats\_persistent, innodb\_stats\_auto\_recalc, and innodb\_stats\_persistent\_sample\_pages are global variables. To override these system- wide settings and configure optimizer statistics parameters for individual tables, you can define STATS\_PERSISTENT, STATS\_AUTO\_RECALC, and STATS\_SAMPLE\_PAGES clauses in CREATE TABLE or ALTER TABLE statements.

• STATS\_PERSISTENT specifies whether to enable persistent statistics for an InnoDB table. The value DEFAULT causes the persistent statistics setting for the table to be determined by the innodb\_stats\_persistent setting. A value of 1 enables persistent statistics for the table, while a value of 0 disables the feature. After enabling persistent statistics for an individual table, use ANALYZE TABLE to calculate statistics after table data is loaded.

• STATS\_AUTO\_RECALC specifies whether to automatically recalculate persistent statistics. The value DEFAULT causes the persistent statistics setting for the table to be determined by the

innodb\_stats\_auto\_recalc setting. A value of 1 causes statistics to be recalculated when 10% of table data has changed. A value 0 prevents automatic recalculation for the table. When using a value of 0, use ANALYZE TABLE to recalculate statistics after making substantial changes to the table.

• STATS\_SAMPLE\_PAGES specifies the number of index pages to sample when cardinality and other statistics are calculated for an indexed column, by an ANALYZE TABLE operation, for example.

All three clauses are specified in the following CREATE TABLE example:

CREATE TABLE `t1` (

`id` int(8) NOT NULL auto\_increment,

`data` varchar(255),

`date` datetime,

PRIMARY KEY (`id`),

INDEX `DATE\_IX` (`date`)

) ENGINE=InnoDB,

STATS\_PERSISTENT=1,

STATS\_AUTO\_RECALC=1,

STATS\_SAMPLE\_PAGES=25;

**Configuring** **the** **Number** **of** **Sampled** **Pages** **for** **InnoDB** **Optimizer** **Statistics**

The optimizer uses estimated statistics about key distributions to choose the indexes for an execution plan, based on the relative selectivity of the index. Operations such as ANALYZE TABLE cause InnoDB to sample random pages from each index on a table to estimate the cardinality of the index. This sampling technique is known as a random dive.

The innodb\_stats\_persistent\_sample\_pages controls the number of sampled pages. You can adjust the setting at runtime to manage the quality of statistics estimates used by the optimizer. The default value is 20. Consider modifying the setting when encountering the following issues:

1. *Statistics* *are* *not* *accurate* *enough* *and* *the* *optimizer* *chooses* *suboptimal* *plans*, as shown in EXPLAIN output. You can check the accuracy of statistics by comparing the actual cardinality of an

index (determined by running SELECT DISTINCT on the index columns) with the estimates in the mysql.innodb\_index\_stats table.

If it is determined that statistics are not accurate enough, the value of innodb\_stats\_persistent\_sample\_pages should be increased until the statistics estimates are sufficiently accurate. Increasing innodb\_stats\_persistent\_sample\_pages too much, however, could cause ANALYZE TABLE to run slowly.

2. *ANALYZE* *TABLE* *is* *too* *slow*. In this case innodb\_stats\_persistent\_sample\_pages should be decreased until ANALYZE TABLE execution time is acceptable. Decreasing the value too much, however, could lead to the first problem of inaccurate statistics and suboptimal query execution plans.

If a balance cannot be achieved between accurate statistics and ANALYZE TABLE execution time, consider decreasing the number of indexed columns in the table or limiting the number of partitions to reduce ANALYZE TABLE complexity. The number of columns in the table's primary key is also important to consider, as primary key columns are appended to each nonunique index.

For related information, see Section 15.8.10.3, “Estimating ANALYZE TABLE Complexity for

InnoDB Tables” .

**Including** **Delete-marked** **Records** **in** **Persistent** **Statistics** **Calculations**

By default, InnoDB reads uncommitted data when calculating statistics. In the case of an uncommitted transaction that deletes rows from a table, delete-marked records are excluded when calculating row estimates and index statistics, which can lead to non-optimal execution plans for other transactions that are operating on the table concurrently using a transaction isolation level other than READ UNCOMMITTED. To avoid this scenario, innodb\_stats\_include\_delete\_marked can be enabled to ensure that delete-marked records are included when calculating persistent optimizer statistics.

When innodb\_stats\_include\_delete\_marked is enabled, ANALYZE TABLE considers delete- marked records when recalculating statistics.

innodb\_stats\_include\_delete\_marked is a global setting that affects all InnoDB tables, and it is only applicable to persistent optimizer statistics.

**InnoDB** **Persistent** **Statistics** **Tables**

The persistent statistics feature relies on the internally managed tables in the mysql database, named innodb\_table\_stats and innodb\_index\_stats. These tables are set up automatically in all install, upgrade, and build-from-source procedures.

**Table** **15.6** **Columns** **of** **innodb\_table\_stats**

|  |  |
| --- | --- |
| **Column** **name** | **Description** |
| database\_name | Database name |
| table\_name | Table name, partition name, or subpartition name |
| last\_update | A timestamp indicating the last time that InnoDB updated this row |
| n\_rows | The number of rows in the table |
| clustered\_index\_size | The size of the primary index, in pages |
| sum\_of\_other\_index\_sizes | The total size of other (non-primary) indexes, in pages |

**Table** **15.7** **Columns** **of** **innodb\_index\_stats**

|  |  |
| --- | --- |
| **Column** **name** | **Description** |
| database\_name | Database name |

|  |  |
| --- | --- |
| **Column** **name** | **Description** |
| table\_name | Table name, partition name, or subpartition name |
| index\_name | Index name |
| last\_update | A timestamp indicating the last time the row was updated |
| stat\_name | The name of the statistic, whose value is reported in the stat\_value column |
| stat\_value | The value of the statistic that is named in stat\_name column |
| sample\_size | The number of pages sampled for the estimate provided in the stat\_value column |
| stat\_description | Description of the statistic that is named in the stat\_name column |

The innodb\_table\_stats and innodb\_index\_stats tables include a last\_update column that shows when index statistics were last updated:

mysql> **SELECT** **\*** **FROM** **innodb\_table\_stats** **\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database\_name: sakila

table\_name: actor

last\_update: 2014-05-28 16:16:44

n\_rows: 200

clustered\_index\_size: 1

sum\_of\_other\_index\_sizes: 1

...

mysql> **SELECT** **\*** **FROM** **innodb\_index\_stats** **\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database\_name: sakila

table\_name: actor

index\_name: PRIMARY

last\_update: 2014-05-28 16:16:44

stat\_name: n\_diff\_pfx01

stat\_value: 200

sample\_size: 1

...

The innodb\_table\_stats and innodb\_index\_stats tables can be updated manually, which makes it possible to force a specific query optimization plan or test alternative plans without modifying the database. If you manually update statistics, use the FLUSH TABLE *tbl\_name* statement to load the updated statistics.

Persistent statistics are considered local information, because they relate to the server instance. The innodb\_table\_stats and innodb\_index\_stats tables are therefore not replicated when automatic statistics recalculation takes place. If you run ANALYZE TABLE to initiate a synchronous recalculation of statistics, the statement is replicated (unless you suppressed logging for it), and recalculation takes place on replicas.

**InnoDB** **Persistent** **Statistics** **Tables** **Example**

The innodb\_table\_stats table contains one row for each table. The following example demonstrates the type of data collected.

Table t1 contains a primary index (columns a, b) secondary index (columns c, d), and unique index (columns e, f):

CREATE TABLE t1 (

a INT, b INT, c INT, d INT, e INT, f INT,

PRIMARY KEY (a, b), KEY i1 (c, d), UNIQUE KEY i2uniq (e, f)

) ENGINE=INNODB;

After inserting five rows of sample data, table t1 appears as follows:

mysql> **SELECT** **\*** **FROM** **t1;**

+---+---+------+------+------+------+

| a | b | c | d | e | f |

+---+---+------+------+------+------+

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | 1  1  1  1  1 | |  |  |  |  | | 1  2  3  4  5 | |  |  |  |  | | 10  10  10  10  10 | |  |  |  |  | | 11  11  11  12  12 | |  |  |  |  | | 100  200  100  200  100 | |  |  |  |  | | 101  102  103  104  105 | |  |  |  |  | |

+---+---+------+------+------+------+

To immediately update statistics, run ANALYZE TABLE (if innodb\_stats\_auto\_recalc is enabled, statistics are updated automatically within a few seconds assuming that the 10% threshold for changed table rows is reached):

mysql> **ANALYZE** **TABLE** **t1;**

+---------+---------+----------+----------+

| Table | Op | Msg\_type | Msg\_text |

+---------+---------+----------+----------+

| test .t1 | analyze | status | OK |

+---------+---------+----------+----------+

Table statistics for table t1 show the last time InnoDB updated the table statistics (2014-03-14 14:36:34), the number of rows in the table (5), the clustered index size (1 page), and the combined size of the other indexes (2 pages).

mysql> **SELECT** **\*** **FROM** **mysql** **.innodb\_table\_stats** **WHERE** **table\_name** **like** **'t1'\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

database\_name: test

table\_name: t1

last\_update: 2014-03-14 14:36:34

n\_rows: 5

clustered\_index\_size: 1

sum\_of\_other\_index\_sizes: 2

The innodb\_index\_stats table contains multiple rows for each index. Each row in the innodb\_index\_stats table provides data related to a particular index statistic which is named in the stat\_name column and described in the stat\_description column. For example:

mysql> **SELECT** **index\_name,** **stat\_name,** **stat\_value,** **stat\_description**

**FROM** **mysql** **.innodb\_index\_stats** **WHERE** **table\_name** **like** **'t1';**

+------------+--------------+------------+-----------------------------------+

| index\_name | stat\_name | stat\_value | stat\_description |

+------------+--------------+------------+-----------------------------------+

|

|

|

|

|

|

|

|

|

|

|

|

|

|

+------------+--------------+------------+-----------------------------------+

The stat\_name column shows the following types of statistics:

|

|

leaf pages in the index |

pages in the index |

|

|

|

|

leaf pages in the index |

pages in the index |

|

|

leaf pages in the index |

pages in the index |

n\_diff\_pfx01

n\_diff\_pfx02

n\_leaf\_pages

size

n\_diff\_pfx01

n\_diff\_pfx02

n\_diff\_pfx03

n\_diff\_pfx04

n\_leaf\_pages

size

n\_diff\_pfx01

n\_diff\_pfx02

n\_leaf\_pages

size

PRIMARY

PRIMARY

PRIMARY

PRIMARY

i1

i1

i1

i1

i1

i1

i2uniq

i2uniq

i2uniq

i2uniq

a

a,b

Number of

Number of

c

c,d

c,d,a

c,d,a,b

Number

Number

e

e,f

Number

Number

1

5

1

1

1

2

2

5

1

1

2

5

1

1

of

of

of

of

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

|

• size: Where stat\_name=size, the stat\_value column displays the total number of pages in the index.

• n\_leaf\_pages: Where stat\_name=n\_leaf\_pages, the stat\_value column displays the number of leaf pages in the index.



• n\_diff\_pfx*NN*: Where stat\_name=n\_diff\_pfx01, the stat\_value column displays the number of distinct values in the first column of the index. Where stat\_name=n\_diff\_pfx02, the stat\_value column displays the number of distinct values in the first two columns of the index, and so on. Where stat\_name=n\_diff\_pfx*NN*, the stat\_description column shows a comma separated list of the index columns that are counted.

To further illustrate the n\_diff\_pfx*NN*statistic, which provides cardinality data, consider once again the t1 table example that was introduced previously. As shown below, the t1 table is created with a primary index (columns a, b), a secondary index (columns c, d), and a unique index (columns e, f):

CREATE TABLE t1 (

a INT, b INT, c INT, d INT, e INT, f INT,

PRIMARY KEY (a, b), KEY i1 (c, d), UNIQUE KEY i2uniq (e, f)

) ENGINE=INNODB;

After inserting five rows of sample data, table t1 appears as follows:

mysql> **SELECT** **\*** **FROM** **t1;**

+---+---+------+------+------+------+

| a | b | c | d | e | f |

+---+---+------+------+------+------+

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | 1  1  1  1  1 | |  |  |  |  | | 1  2  3  4  5 | |  |  |  |  | | 10  10  10  10  10 | |  |  |  |  | | 11  11  11  12  12 | |  |  |  |  | | 100  200  100  200  100 | |  |  |  |  | | 101  102  103  104  105 | |  |  |  |  | |

+---+---+------+------+------+------+

When you query the index\_name, stat\_name, stat\_value, and stat\_description, where stat\_name LIKE 'n\_diff%', the following result set is returned:

mysql> **SELECT** **index\_name,** **stat\_name,** **stat\_value,** **stat\_description**

**FROM** **mysql.innodb\_index\_stats**

**WHERE** **table\_name** **like** **'t1'** **AND** **stat\_name** **LIKE** **'n\_diff%';**

+------------+--------------+------------+------------------+

| index\_name | stat\_name | stat\_value | stat\_description |

+------------+--------------+------------+------------------+

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| | | PRIMARY | | | n\_diff\_pfx01 | | 1 | | | a | | |
| | | PRIMARY | | | n\_diff\_pfx02 | | 5 | | | a,b | | |
| | | i1 | | | n\_diff\_pfx01 | | 1 | | | c | | |
| | | i1 | | | n\_diff\_pfx02 | | 2 | | | c,d | | |
| | | i1 | | | n\_diff\_pfx03 | | 2 | | | c,d,a | | |
| | | i1 | | | n\_diff\_pfx04 | | 5 | | | c,d,a,b | | |
| | | i2uniq | | | n\_diff\_pfx01 | | 2 | | | e | | |
| | | i2uniq | | | n\_diff\_pfx02 | | 5 | | | e,f | | |

+------------+--------------+------------+------------------+

For the PRIMARY index, there are two n\_diff% rows. The number of rows is equal to the number of columns in the index.

**Note**

For nonunique indexes, InnoDB appends the columns of the primary key.

• Where index\_name=PRIMARY and stat\_name=n\_diff\_pfx01, the stat\_value is 1, which indicates that there is a single distinct value in the first column of the index (column a). The number of distinct values in column a is confirmed by viewing the data in column a in table t1, in which there is a single distinct value (1). The counted column (a) is shown in the stat\_description column of the result set.

• Where index\_name=PRIMARY and stat\_name=n\_diff\_pfx02, the stat\_value is 5, which indicates that there are five distinct values in the two columns of the index (a,b). The number of distinct values in columns a and b is confirmed by viewing the data in columns a and b in table t1, in which there are five distinct values: (1,1), (1,2), (1,3), (1,4) and (1,5). The counted columns (a,b) are shown in the stat\_description column of the result set.

For the secondary index (i1), there are four n\_diff% rows. Only two columns are defined for the secondary index (c,d) but there are four n\_diff% rows for the secondary index because InnoDB suffixes all nonunique indexes with the primary key. As a result, there are four n\_diff% rows instead of two to account for the both the secondary index columns (c,d) and the primary key columns (a,b).

• Where index\_name=i1 and stat\_name=n\_diff\_pfx01, the stat\_value is 1, which indicates that there is a single distinct value in the first column of the index (column c). The number of distinct values in column c is confirmed by viewing the data in column c in table t1, in which there is a single distinct value: (10). The counted column (c) is shown in the stat\_description column of the result set.

• Where index\_name=i1 and stat\_name=n\_diff\_pfx02, the stat\_value is 2, which indicates that there are two distinct values in the first two columns of the index (c,d). The number of distinct values in columns c an d is confirmed by viewing the data in columns c and d in table t1, in which there are two distinct values: (10,11) and (10,12). The counted columns (c,d) are shown in the stat\_description column of the result set.

• Where index\_name=i1 and stat\_name=n\_diff\_pfx03, the stat\_value is 2, which indicates that there are two distinct values in the first three columns of the index (c,d,a). The number of distinct values in columns c, d, and a is confirmed by viewing the data in column c, d, and a in table t1, in which there are two distinct values: (10,11,1) and (10,12,1). The counted columns (c,d,a) are shown in the stat\_description column of the result set.

• Where index\_name=i1 and stat\_name=n\_diff\_pfx04, the stat\_value is 5, which indicates that there are five distinct values in the four columns of the index (c,d,a,b). The number of distinct values in columns c, d, a and b is confirmed by viewing the data in columns

c, d, a, and b in table t1, in which there are five distinct values: ( 10,11,1,1), ( 10,11,1,2), ( 10,11,1,3), ( 10,12,1,4), and ( 10,12,1,5). The counted columns (c,d,a,b) are shown in the stat\_description column of the result set.

For the unique index (i2uniq), there are two n\_diff% rows.

• Where index\_name=i2uniq and stat\_name=n\_diff\_pfx01, the stat\_value is 2, which indicates that there are two distinct values in the first column of the index (column e). The number of distinct values in column e is confirmed by viewing the data in column e in table t1, in which there are two distinct values: (100) and (200). The counted column (e) is shown in the stat\_description column of the result set.

• Where index\_name=i2uniq and stat\_name=n\_diff\_pfx02, the stat\_value is 5, which indicates that there are five distinct values in the two columns of the index (e,f). The number of distinct values in columns e and f is confirmed by viewing the data in columns e and f in table t1, in which there are five distinct values: (100,101), (200,102), (100,103), (200,104), and (100,105). The counted columns (e,f) are shown in the stat\_description column of the result set.

**Retrieving** **Index** **Size** **Using** **the** **innodb\_index\_stats** **Table**

You can retrieve the index size for tables, partitions, or subpartitions can using the innodb\_index\_stats table. In the following example, index sizes are retrieved for table t1. For a definition of table t1 and corresponding index statistics, see [InnoDB Persistent Statistics Tables](#_bookmark33) [Example](#_bookmark33).

mysql> **SELECT** **SUM(stat\_value)** **pages,** **index\_name,**

**SUM(stat\_value)\*@@innodb\_page\_size** **size**

**FROM** **mysql.innodb\_index\_stats** **WHERE** **table\_name='t1'**

**AND** **stat\_name** **=** **'size'** **GROUP** **BY** **index\_name;**

+-------+------------+-------+

| pages | index\_name | size |

+-------+------------+-------+

| 1 | PRIMARY | 16384 |

| 1 | i1 | 16384 |

| 1 | i2uniq | 16384 |

+-------+------------+-------+