• “Bulk inserts”

Statements for which the number of rows to be inserted (and the number of required auto- increment values) is not known in advance. This includes INSERT ... SELECT, REPLACE ... SELECT, and LOAD DATA statements, but not plain INSERT. InnoDB assigns new values for the AUTO\_INCREMENT column one at a time as each row is processed.

• “Mixed-mode inserts”

These are “simple insert” statements that specify the auto-increment value for some (but not all) of the new rows. An example follows, where c1 is an AUTO\_INCREMENT column of table t1:

INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d');

Another type of “mixed-mode insert” is INSERT ... ON DUPLICATE KEY UPDATE, which in the worst case is in effect an INSERT followed by a UPDATE, where the allocated value for the AUTO\_INCREMENT column may or may not be used during the update phase.

There are three possible settings for the innodb\_autoinc\_lock\_mode variable. The settings are 0, 1, or 2, for “traditional” , “consecutive” , or “interleaved” lock mode, respectively. As of MySQL 8.0, interleaved lock mode (innodb\_autoinc\_lock\_mode=2) is the default setting. Prior to MySQL 8.0, consecutive lock mode is the default (innodb\_autoinc\_lock\_mode=1).

The default setting of interleaved lock mode in MySQL 8.0 reflects the change from statement-based replication to row based replication as the default replication type. Statement-based replication requires the consecutive auto-increment lock mode to ensure that auto-increment values are assigned in a predictable and repeatable order for a given sequence of SQL statements, whereas row-based replication is not sensitive to the execution order of SQL statements.

• innodb\_autoinc\_lock\_mode = 0 (“traditional” lock mode)

The traditional lock mode provides the same behavior that existed before the innodb\_autoinc\_lock\_mode variable was introduced. The traditional lock mode option is provided for backward compatibility, performance testing, and working around issues with “mixed- mode inserts”, due to possible differences in semantics.

In this lock mode, all “INSERT-like” statements obtain a special table-level AUTO-INC lock for inserts into tables with AUTO\_INCREMENT columns. This lock is normally held to the end of the statement (not to the end of the transaction) to ensure that auto-increment values are assigned in a predictable and repeatable order for a given sequence of INSERT statements, and to ensure that auto-increment values assigned by any given statement are consecutive.

In the case of statement-based replication, this means that when an SQL statement is replicated on a replica server, the same values are used for the auto-increment column as on the source server. The result of execution of multiple INSERT statements is deterministic, and the replica reproduces the same data as on the source. If auto-increment values generated by multiple INSERT statements were interleaved, the result of two concurrent INSERT statements would be nondeterministic, and could not reliably be propagated to a replica server using statement-based replication.

To make this clear, consider an example that uses this table:

CREATE TABLE t1 (

c1 INT(11) NOT NULL AUTO\_INCREMENT,

c2 VARCHAR(10) DEFAULT NULL,

PRIMARY KEY (c1)

) ENGINE=InnoDB;

Suppose that there are two transactions running, each inserting rows into a table with an AUTO\_INCREMENT column. One transaction is using an INSERT ... SELECT statement that inserts 1000 rows, and another is using a simple INSERT statement that inserts one row:

Tx1: INSERT INTO t1 (c2) SELECT 1000 rows from another table ...

Tx2: INSERT INTO t1 (c2) VALUES ('xxx');

InnoDB cannot tell in advance how many rows are retrieved from the SELECT in the INSERT statement in Tx1, and it assigns the auto-increment values one at a time as the statement proceeds. With a table-level lock, held to the end of the statement, only one INSERT statement referring to table t1 can execute at a time, and the generation of auto-increment numbers by different statements is not interleaved. The auto-increment values generated by the Tx1 INSERT ... SELECT statement are consecutive, and the (single) auto-increment value used by the INSERT statement in Tx2 is either smaller or larger than all those used for Tx1, depending on which statement executes first.

As long as the SQL statements execute in the same order when replayed from the binary log (when using statement-based replication, or in recovery scenarios), the results are the same as they were when Tx1 and Tx2 first ran. Thus, table-level locks held until the end of a statement make INSERT statements using auto-increment safe for use with statement-based replication. However, those table-level locks limit concurrency and scalability when multiple transactions are executing insert statements at the same time.

In the preceding example, if there were no table-level lock, the value of the auto-increment column used for the INSERT in Tx2 depends on precisely when the statement executes. If the INSERT of Tx2 executes while the INSERT of Tx1 is running (rather than before it starts or after it completes), the specific auto-increment values assigned by the two INSERT statements are nondeterministic, and may vary from run to run.

Under the [consecutive](#_bookmark1) lock mode, InnoDB can avoid using table-level AUTO-INC locks for “simple insert” statements where the number of rows is known in advance, and still preserve deterministic execution and safety for statement-based replication.

If you are not using the binary log to replay SQL statements as part of recovery or replication, the [interleaved](#_bookmark2) lock mode can be used to eliminate all use of table-level AUTO-INC locks for even greater concurrency and performance, at the cost of permitting gaps in auto-increment numbers assigned by a statement and potentially having the numbers assigned by concurrently executing statements interleaved.

• innodb\_autoinc\_lock\_mode = 1 (“consecutive” lock mode)

In this mode, “bulk inserts” use the special AUTO-INC table-level lock and hold it until the end of the statement. This applies to all INSERT ... SELECT, REPLACE ... SELECT, and LOAD DATA statements. Only one statement holding the AUTO-INC lock can execute at a time. If the source table of the bulk insert operation is different from the target table, the AUTO-INC lock on the target table is taken after a shared lock is taken on the first row selected from the source table. If the source and target of the bulk insert operation are the same table, the AUTO-INC lock is taken after shared locks are taken on all selected rows.

“Simple inserts” (for which the number of rows to be inserted is known in advance) avoid table-level AUTO-INC locks by obtaining the required number of auto-increment values under the control of a mutex (a light-weight lock) that is only held for the duration of the allocation process, *not* until the statement completes. No table-level AUTO-INC lock is used unless an AUTO-INC lock is held by another transaction. If another transaction holds an AUTO-INC lock, a “simple insert” waits for the AUTO-INC lock, as if it were a “bulk insert” .

This lock mode ensures that, in the presence of INSERT statements where the number of rows is not known in advance (and where auto-increment numbers are assigned as the statement progresses), all auto-increment values assigned by any “INSERT-like” statement are consecutive, and operations are safe for statement-based replication.

Simply put, this lock mode significantly improves scalability while being safe for use with statement- based replication. Further, as with “traditional” lock mode, auto-increment numbers assigned by any given statement are *consecutive*. There is *no* *change* in semantics compared to “traditional” mode for any statement that uses auto-increment, with one important exception.

The exception is for “mixed-mode inserts” , where the user provides explicit values for an AUTO\_INCREMENT column for some, but not all, rows in a multiple-row “simple insert” . For such inserts, InnoDB allocates more auto-increment values than the number of rows to be inserted. However, all values automatically assigned are consecutively generated (and thus higher than) the auto-increment value generated by the most recently executed previous statement. “Excess” numbers are lost.

• innodb\_autoinc\_lock\_mode = 2 (“interleaved” lock mode)

In this lock mode, no “INSERT-like” statements use the table-level AUTO-INC lock, and multiple statements can execute at the same time. This is the fastest and most scalable lock mode, but it is *not* *safe* when using statement-based replication or recovery scenarios when SQL statements are replayed from the binary log.

In this lock mode, auto-increment values are guaranteed to be unique and monotonically increasing across all concurrently executing “INSERT-like” statements. However, because multiple statements can be generating numbers at the same time (that is, allocation of numbers is *interleaved* across statements), the values generated for the rows inserted by any given statement may not be consecutive.

If the only statements executing are “simple inserts” where the number of rows to be inserted is known ahead of time, there are no gaps in the numbers generated for a single statement, except for “mixed-mode inserts” . However, when “bulk inserts” are executed, there may be gaps in the auto- increment values assigned by any given statement.

**InnoDB** **AUTO\_INCREMENT** **Lock** **Mode** **Usage** **Implications**

• Using auto-increment with replication

If you are using statement-based replication, set innodb\_autoinc\_lock\_mode to 0 or 1 and use the same value on the source and its replicas. Auto-increment values are not ensured to be the same on the replicas as on the source if you use innodb\_autoinc\_lock\_mode = 2 (“interleaved”) or configurations where the source and replicas do not use the same lock mode.

If you are using row-based or mixed-format replication, all of the auto-increment lock modes are safe, since row-based replication is not sensitive to the order of execution of the SQL statements (and the mixed format uses row-based replication for any statements that are unsafe for statement-based replication).

• “Lost” auto-increment values and sequence gaps

In all lock modes (0, 1, and 2), if a transaction that generated auto-increment values rolls back, those auto-increment values are “lost” . Once a value is generated for an auto-increment column, it cannot be rolled back, whether or not the “INSERT-like” statement is completed, and whether or not the containing transaction is rolled back. Such lost values are not reused. Thus, there may be gaps in the values stored in an AUTO\_INCREMENT column of a table.

• Specifying NULL or 0 for the AUTO\_INCREMENT column

In all lock modes (0, 1, and 2), if a user specifies NULL or 0 for the AUTO\_INCREMENT column in an INSERT, InnoDB treats the row as if the value was not specified and generates a new value for it.

• Assigning a negative value to the AUTO\_INCREMENT column

In all lock modes (0, 1, and 2), the behavior of the auto-increment mechanism is undefined if you assign a negative value to the AUTO\_INCREMENT column.

• If the AUTO\_INCREMENT value becomes larger than the maximum integer for the specified integer type

In all lock modes (0, 1, and 2), the behavior of the auto-increment mechanism is undefined if the value becomes larger than the maximum integer that can be stored in the specified integer type.

• Gaps in auto-increment values for “bulk inserts”

With innodb\_autoinc\_lock\_mode set to 0 (“traditional”) or 1 (“consecutive”), the auto-increment values generated by any given statement are consecutive, without gaps, because the table-level AUTO-INC lock is held until the end of the statement, and only one such statement can execute at a time.

With innodb\_autoinc\_lock\_mode set to 2 (“interleaved”), there may be gaps in the auto- increment values generated by “bulk inserts,” but only if there are concurrently executing “INSERT- like” statements.

For lock modes 1 or 2, gaps may occur between successive statements because for bulk inserts the exact number of auto-increment values required by each statement may not be known and overestimation is possible.

• Auto-increment values assigned by “mixed-mode inserts”

Consider a “mixed-mode insert,” where a “simple insert” specifies the auto-increment value for some (but not all) resulting rows. Such a statement behaves differently in lock modes 0, 1, and 2. For example, assume c1 is an AUTO\_INCREMENT column of table t1, and that the most recent automatically generated sequence number is 100.

mysql> **CREATE** **TABLE** **t1** **(**

-> **c1** **INT** **UNSIGNED** **NOT** **NULL** **AUTO\_INCREMENT** **PRIMARY** **KEY,**

-> **c2** **CHAR(1)**

-> **)** **ENGINE** **=** **INNODB;**

Now, consider the following “mixed-mode insert” statement:

mysql> **INSERT** **INTO** **t1** **(c1,c2)** **VALUES** **(1,'a'),** **(NULL,'b'),** **(5,'c'),** **(NULL,'d');**

With innodb\_autoinc\_lock\_mode set to 0 (“traditional”), the four new rows are:

mysql> **SELECT** **c1,** **c2** **FROM** **t1** **ORDER** **BY** **c2;**

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

| c1 | c2 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | 1  101  5  102 | |  |  |  | | a  b  c  d | |  |  |  | |

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

The next available auto-increment value is 103 because the auto-increment values are allocated one at a time, not all at once at the beginning of statement execution. This result is true whether or not there are concurrently executing “INSERT-like” statements (of any type).

With innodb\_autoinc\_lock\_mode set to 1 (“consecutive”), the four new rows are also:

mysql> **SELECT** **c1,** **c2** **FROM** **t1** **ORDER** **BY** **c2;**

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

| c1 | c2 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | 1  101  5  102 | |  |  |  | | a  b  c  d | |  |  |  | |

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

However, in this case, the next available auto-increment value is 105, not 103 because four auto- increment values are allocated at the time the statement is processed, but only two are used. This result is true whether or not there are concurrently executing “INSERT-like” statements (of any type).

With innodb\_autoinc\_lock\_mode set to 2 (“interleaved”), the four new rows are:

mysql> **SELECT** **c1,** **c2** **FROM** **t1** **ORDER** **BY** **c2;**

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

| c1 | c2 |

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

|  |  |  |
| --- | --- | --- |
| |  |  |  | | 1 | a  *x* | b  5 | c  *y* | d | |  |  |  | |

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

The values of *x* and *y* are unique and larger than any previously generated rows. However, the specific values of *x* and *y* depend on the number of auto-increment values generated by concurrently executing statements.

Finally, consider the following statement, issued when the most-recently generated sequence number is 100:

mysql> **INSERT** **INTO** **t1** **(c1,c2)** **VALUES** **(1,'a'),** **(NULL,'b'),** **(101,'c'),** **(NULL,'d');**

With any innodb\_autoinc\_lock\_mode setting, this statement generates a duplicate-key error

23000 (Can't write; duplicate key in table) because 101 is allocated for the row (NULL, 'b') and insertion of the row (101, 'c') fails.

• Modifying AUTO\_INCREMENT column values in the middle of a sequence of INSERT statements

In MySQL 5.7 and earlier, modifying an AUTO\_INCREMENT column value in the middle of a sequence of INSERT statements could lead to “Duplicate entry” errors. For example, if you performed an UPDATE operation that changed an AUTO\_INCREMENT column value to a value larger than the current maximum auto-increment value, subsequent INSERT operations that did not specify an unused auto-increment value could encounter “Duplicate entry” errors. In MySQL 8.0 and later, if you modify an AUTO\_INCREMENT column value to a value larger than the current maximum auto- increment value, the new value is persisted, and subsequent INSERT operations allocate auto- increment values starting from the new, larger value. This behavior is demonstrated in the following example.

mysql> **CREATE** **TABLE** **t1** **(**

-> **c1** **INT** **NOT** **NULL** **AUTO\_INCREMENT,**

-> **PRIMARY** **KEY** **(c1)**

-> **)** **ENGINE** **=** **InnoDB;**

mysql> **INSERT** **INTO** **t1** **VALUES(0),** **(0),** **(3);**

mysql> **SELECT** **c1** **FROM** **t1;**

+----+

| c1 |

+----+

| 1 |

| 2 |

| 3 |

+----+

mysql> **UPDATE** **t1** **SET** **c1** **=** **4** **WHERE** **c1** **=** **1;**

mysql> **SELECT** **c1** **FROM** **t1;**

+----+

| c1 |

+----+

| 2 |