| 3 |

| 4 |

+----+

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

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

+----+

| c1 |

+----+

|  |  |  |
| --- | --- | --- |
| |  |  |  | | 2  3  4  5 | |  |  |  | |

+----+

**InnoDB** **AUTO\_INCREMENT** **Counter** **Initialization**

This section describes how InnoDB initializes AUTO\_INCREMENT counters.

If you specify an AUTO\_INCREMENT column for an InnoDB table, the in-memory table object contains a special counter called the auto-increment counter that is used when assigning new values for the column.

In MySQL 5.7 and earlier, the auto-increment counter is stored in main memory, not on disk. To initialize an auto-increment counter after a server restart, InnoDB would execute the equivalent of the following statement on the first insert into a table containing an AUTO\_INCREMENT column.

SELECT MAX(ai\_col) FROM *table\_name* FOR UPDATE;

In MySQL 8.0, this behavior is changed. The current maximum auto-increment counter value is written to the redo log each time it changes and saved to the data dictionary on each checkpoint. These changes make the current maximum auto-increment counter value persistent across server restarts.

On a server restart following a normal shutdown, InnoDB initializes the in-memory auto-increment counter using the current maximum auto-increment value stored in the data dictionary.

On a server restart during crash recovery, InnoDB initializes the in-memory auto-increment counter using the current maximum auto-increment value stored in the data dictionary and scans the redo log for auto-increment counter values written since the last checkpoint. If a redo-logged value is greater than the in-memory counter value, the redo-logged value is applied. However, in the case of an unexpected server exit, reuse of a previously allocated auto-increment value cannot be guaranteed. Each time the current maximum auto-increment value is changed due to an INSERT or UPDATE operation, the new value is written to the redo log, but if the unexpected exit occurs before the redo log is flushed to disk, the previously allocated value could be reused when the auto-increment counter is initialized after the server is restarted.

The only circumstance in which InnoDB uses the equivalent of a SELECT MAX(ai\_col) FROM *table\_name* FOR UPDATE statement to initialize an auto-increment counter is when importing a table without a .cfg metadata file. Otherwise, the current maximum auto-increment counter value is read from the .cfg metadata file if present. Aside from counter value initialization, the equivalent of a SELECT MAX(ai\_col) FROM *table\_name* statement is used to determine the current maximum auto-increment counter value of the table when attempting to set the counter value to one that is smaller than or equal to the persisted counter value using an ALTER TABLE ... AUTO\_INCREMENT = *N*statement. For example, you might try to set the counter value to a lesser value after deleting some records. In this case, the table must be searched to ensure that the new counter value is not less than or equal to the actual current maximum counter value.

In MySQL 5.7 and earlier, a server restart cancels the effect of the AUTO\_INCREMENT = N table option, which may be used in a CREATE TABLE or ALTER TABLE statement to set an initial counter value or alter the existing counter value, respectively. In MySQL 8.0, a server restart does not cancel the effect of the AUTO\_INCREMENT = N table option. If you initialize the auto-increment counter to



a specific value, or if you alter the auto-increment counter value to a larger value, the new value is persisted across server restarts.

**Note**

ALTER TABLE ... AUTO\_INCREMENT = N can only change the auto- increment counter value to a value larger than the current maximum.

In MySQL 5.7 and earlier, a server restart immediately following a ROLLBACK operation could result in the reuse of auto-increment values that were previously allocated to the rolled-back transaction, effectively rolling back the current maximum auto-increment value. In MySQL 8.0, the current maximum auto-increment value is persisted, preventing the reuse of previously allocated values.

If a SHOW TABLE STATUS statement examines a table before the auto-increment counter is initialized, InnoDB opens the table and initializes the counter value using the current maximum auto-increment value that is stored in the data dictionary. The value is then stored in memory for use by later inserts or updates. Initialization of the counter value uses a normal exclusive-locking read on the table which lasts to the end of the transaction. InnoDB follows the same procedure when initializing the auto- increment counter for a newly created table that has a user-specified auto-increment value greater than

0.

After the auto-increment counter is initialized, if you do not explicitly specify an auto-increment value when inserting a row, InnoDB implicitly increments the counter and assigns the new value to the column. If you insert a row that explicitly specifies an auto-increment column value, and the value is greater than the current maximum counter value, the counter is set to the specified value.

InnoDB uses the in-memory auto-increment counter as long as the server runs. When the server is stopped and restarted, InnoDB reinitializes the auto-increment counter, as described earlier.

The auto\_increment\_offset variable determines the starting point for the AUTO\_INCREMENT column value. The default setting is 1.

The auto\_increment\_increment variable controls the interval between successive column values. The default setting is 1.

**Notes**

When an AUTO\_INCREMENT integer column runs out of values, a subsequent INSERT operation returns a duplicate-key error. This is general MySQL behavior.

**15.6.2** **Indexes**

This section covers topics related to InnoDB indexes.

**15.6.2.1** **Clustered** **and** **Secondary** **Indexes**

Each InnoDB table has a special index called the clustered index that stores row data. Typically, the clustered index is synonymous with the primary key. To get the best performance from queries, inserts, and other database operations, it is important to understand how InnoDB uses the clustered index to optimize the common lookup and DML operations.

• When you define a PRIMARY KEY on a table, InnoDB uses it as the clustered index. A primary key should be defined for each table. If there is no logical unique and non-null column or set of columns to use a the primary key, add an auto-increment column. Auto-increment column values are unique and are added automatically as new rows are inserted.

• If you do not define a PRIMARY KEY for a table, InnoDB uses the first UNIQUE index with all key columns defined as NOT NULL as the clustered index.

• If a table has no PRIMARY KEY or suitable UNIQUE index, InnoDB generates a hidden clustered index named GEN\_CLUST\_INDEX on a synthetic column that contains row ID values. The rows are

ordered by the row ID that InnoDB assigns. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in order of insertion.

**How** **the** **Clustered** **Index** **Speeds** **Up** **Queries**

Accessing a row through the clustered index is fast because the index search leads directly to the page that contains the row data. If a table is large, the clustered index architecture often saves a disk I/O operation when compared to storage organizations that store row data using a different page from the index record.

**How** **Secondary** **Indexes** **Relate** **to** **the** **Clustered** **Index**

Indexes other than the clustered index are known as secondary indexes. In InnoDB, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. InnoDB uses this primary key value to search for the row in the clustered index.

If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.

For guidelines to take advantage of InnoDB clustered and secondary indexes, see Section 8.3, “Optimization and Indexes” .

**15.6.2.2** **The** **Physical** **Structure** **of** **an** **InnoDB** **Index**

With the exception of spatial indexes, InnoDB indexes are B-tree data structures. Spatial indexes use R-trees, which are specialized data structures for indexing multi-dimensional data. Index records are stored in the leaf pages of their B-tree or R-tree data structure. The default size of an index page is 16KB. The page size is determined by the innodb\_page\_size setting when the MySQL instance is initialized. See Section 15.8.1, “InnoDB Startup Configuration” .

When new records are inserted into an InnoDB clustered index, InnoDB tries to leave 1/16 of the page free for future insertions and updates of the index records. If index records are inserted in a sequential order (ascending or descending), the resulting index pages are about 15/16 full. If records are inserted in a random order, the pages are from 1/2 to 15/16 full.

InnoDB performs a bulk load when creating or rebuilding B-tree indexes. This method of index creation is known as a sorted index build. The innodb\_fill\_factor variable defines the percentage of space on each B-tree page that is filled during a sorted index build, with the remaining space reserved for future index growth. Sorted index builds are not supported for spatial indexes. For more information, see [Section 15.6.2.3, “Sorted Index Builds”](#_bookmark2) . An innodb\_fill\_factor setting of 100 leaves 1/16 of the space in clustered index pages free for future index growth.

If the fill factor of an InnoDB index page drops below the MERGE\_THRESHOLD, which is 50% by default if not specified, InnoDB tries to contract the index tree to free the page. The MERGE\_THRESHOLD setting applies to both B-tree and R-tree indexes. For more information, see Section 15.8.11, “Configuring the Merge Threshold for Index Pages” .

**15.6.2.3** **Sorted** **Index** **Builds**

InnoDB performs a bulk load instead of inserting one index record at a time when creating or rebuilding indexes. This method of index creation is also known as a sorted index build. Sorted index builds are not supported for spatial indexes.

There are three phases to an index build. In the first phase, the clustered index is scanned, and index entries are generated and added to the sort buffer. When the sort buffer becomes full, entries are sorted and written out to a temporary intermediate file. This process is also known as a “run” . In the second phase, with one or more runs written to the temporary intermediate file, a merge sort is performed on all entries in the file. In the third and final phase, the sorted entries are inserted into the B-tree.

Prior to the introduction of sorted index builds, index entries were inserted into the B-tree one record at a time using insert APIs. This method involved opening a B-tree cursor to find the insert position and then inserting entries into a B-tree page using an optimistic insert. If an insert failed due to a page being full, a pessimistic insert would be performed, which involves opening a B-tree cursor and splitting and merging B-tree nodes as necessary to find space for the entry. The drawbacks of this “top-down” method of building an index are the cost of searching for an insert position and the constant splitting and merging of B-tree nodes.

Sorted index builds use a “bottom-up” approach to building an index. With this approach, a reference to the right-most leaf page is held at all levels of the B-tree. The right-most leaf page at the necessary B- tree depth is allocated and entries are inserted according to their sorted order. Once a leaf page is full, a node pointer is appended to the parent page and a sibling leaf page is allocated for the next insert. This process continues until all entries are inserted, which may result in inserts up to the root level. When a sibling page is allocated, the reference to the previously pinned leaf page is released, and the newly allocated leaf page becomes the right-most leaf page and new default insert location.

**Reserving** **B-tree** **Page** **Space** **for** **Future** **Index** **Growth**

To set aside space for future index growth, you can use the innodb\_fill\_factor variable to reserve a percentage of B-tree page space. For example, setting innodb\_fill\_factor to 80 reserves 20 percent of the space in B-tree pages during a sorted index build. This setting applies to both B-tree leaf and non-leaf pages. It does not apply to external pages used for TEXT or BLOB entries. The amount of space that is reserved may not be exactly as configured, as the innodb\_fill\_factor value is interpreted as a hint rather than a hard limit.

**Sorted** **Index** **Builds** **and** **Full-Text** **Index** **Support**

Sorted index builds are supported for fulltext indexes. Previously, SQL was used to insert entries into a fulltext index.

**Sorted** **Index** **Builds** **and** **Compressed** **Tables**

For compressed tables, the previous index creation method appended entries to both compressed and uncompressed pages. When the modification log (representing free space on the compressed page) became full, the compressed page would be recompressed. If compression failed due to a lack of space, the page would be split. With sorted index builds, entries are only appended to uncompressed pages. When an uncompressed page becomes full, it is compressed. Adaptive padding is used to ensure that compression succeeds in most cases, but if compression fails, the page is split and compression is attempted again. This process continues until compression is successful. For more information about compression of B-Tree pages, see Section 15.9.1.5, “How Compression Works for

InnoDB Tables” .

**Sorted** **Index** **Builds** **and** **Redo** **Logging**

Redo logging is disabled during a sorted index build. Instead, there is a checkpoint to ensure that the index build can withstand an unexpected exit or failure. The checkpoint forces a write of all dirty pages to disk. During a sorted index build, the page cleaner thread is signaled periodically to flush dirty pages to ensure that the checkpoint operation can be processed quickly. Normally, the page cleaner thread flushes dirty pages when the number of clean pages falls below a set threshold. For sorted index builds, dirty pages are flushed promptly to reduce checkpoint overhead and to parallelize I/O and CPU activity.

**Sorted** **Index** **Builds** **and** **Optimizer** **Statistics**

Sorted index builds may result in optimizer statistics that differ from those generated by the previous method of index creation. The difference in statistics, which is not expected to affect workload performance, is due to the different algorithm used to populate the index.

**15.6.2.4** **InnoDB** **Full-Text** **Indexes**

Full-text indexes are created on text-based columns (CHAR, VARCHAR, or TEXT columns) to speed up queries and DML operations on data contained within those columns.

A full-text index is defined as part of a CREATE TABLE statement or added to an existing table using ALTER TABLE or CREATE INDEX.

Full-text search is performed using MATCH() ... AGAINST syntax. For usage information, see Section 12.10, “Full-Text Search Functions” .

InnoDB full-text indexes are described under the following topics in this section:

• [InnoDB Full-Text Index Design](#_bookmark3)

• [InnoDB Full-Text Index Tables](#_bookmark4)

• [InnoDB Full-Text Index Cache](#_bookmark5)

• [InnoDB Full-Text Index DOC\_ID and FTS\_DOC\_ID Column](#_bookmark6)

• [InnoDB Full-Text Index Deletion Handling](#_bookmark7)

• [InnoDB Full-Text Index Transaction Handling](#_bookmark8)

• [Monitoring InnoDB Full-Text Indexes](#_bookmark9)

**InnoDB** **Full-Text** **Index** **Design**

InnoDB full-text indexes have an inverted index design. Inverted indexes store a list of words, and for each word, a list of documents that the word appears in. To support proximity search, position information for each word is also stored, as a byte offset.

**InnoDB** **Full-Text** **Index** **Tables**

When an InnoDB full-text index is created, a set of index tables is created, as shown in the following example:

mysql> **CREATE** **TABLE** **opening\_lines** **(**

**id** **INT** **UNSIGNED** **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,**

**opening\_line** **TEXT(500),**

**author** **VARCHAR(200),**

**title** **VARCHAR(200),**

**FULLTEXT** **idx** **(opening\_line)**

**)** **ENGINE=InnoDB;**

mysql> **SELECT** **table\_id,** **name,** **space** **from** **INFORMATION\_SCHEMA.INNODB\_TABLES**

**WHERE** **name** **LIKE** **'test/%';**

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

| space |

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | 333  334  335  336  337  338  330  331  332  328  329  327 | |  |  |  |  |  |  |  | | test/fts\_0000000000000147\_00000000000001c9\_index\_1  test/fts\_0000000000000147\_00000000000001c9\_index\_2  test/fts\_0000000000000147\_00000000000001c9\_index\_3  test/fts\_0000000000000147\_00000000000001c9\_index\_4  test/fts\_0000000000000147\_00000000000001c9\_index\_5  test/fts\_0000000000000147\_00000000000001c9\_index\_6  test/fts\_0000000000000147\_being\_deleted  test/fts\_0000000000000147\_being\_deleted\_cache  test/fts\_0000000000000147\_config  test/fts\_0000000000000147\_deleted  test/fts\_0000000000000147\_deleted\_cache  test/opening lines | |  |  |  |  |  |  |  | |  | 289  290  291  292  293  294  286  287  288  284  285  283 | |  |  |  |  |  |  |  | |

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

| table\_id | name

The first six index tables comprise the inverted index and are referred to as auxiliary index tables. When incoming documents are tokenized, the individual words (also referred to as “tokens”) are

inserted into the index tables along with position information and an associated DOC\_ID. The words are fully sorted and partitioned among the six index tables based on the character set sort weight of the word's first character.

The inverted index is partitioned into six auxiliary index tables to support parallel index creation. By default, two threads tokenize, sort, and insert words and associated data into the index tables. The number of threads that perform this work is configurable using the innodb\_ft\_sort\_pll\_degree variable. Consider increasing the number of threads when creating full-text indexes on large tables.

Auxiliary index table names are prefixed with fts\_ and postfixed with index\_*#*. Each auxiliary index table is associated with the indexed table by a hex value in the auxiliary index table name that matches the table\_id of the indexed table. For example, the table\_id of the test/opening\_lines table is 327, for which the hex value is 0x147. As shown in the preceding example, the “147” hex value appears in the names of auxiliary index tables that are associated with the test/opening\_lines table.

A hex value representing the index\_id of the full-text index also appears in

auxiliary index table names. For example, in the auxiliary table name test/ fts\_0000000000000147\_00000000000001c9\_index\_1, the hex value 1c9 has a decimal value of 457. The index defined on the opening\_lines table (idx) can be identified by querying the Information Schema INNODB\_INDEXES table for this value (457).

mysql> **SELECT** **index\_id,** **name,** **table\_id,** **space** **from** **INFORMATION\_SCHEMA.INNODB\_INDEXES**

**WHERE** **index\_id=457;**

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

| index\_id | name | table\_id | space |

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

| 457 | idx | 327 | 283 |

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

Index tables are stored in their own tablespace if the primary table is created in a file-per-table tablespace. Otherwise, index tables are stored in the tablespace where the indexed table resides.

The other index tables shown in the preceding example are referred to as common index tables and are used for deletion handling and storing the internal state of full-text indexes. Unlike the inverted index tables, which are created for each full-text index, this set of tables is common to all full-text indexes created on a particular table.

Common index tables are retained even if full-text indexes are dropped. When a full-text index is dropped, the FTS\_DOC\_ID column that was created for the index is retained, as removing the FTS\_DOC\_ID column would require rebuilding the previously indexed table. Common index tables are required to manage the FTS\_DOC\_ID column.

• fts\_\*\_deleted and fts\_\*\_deleted\_cache

Contain the document IDs (DOC\_ID) for documents that are deleted but whose data is not yet removed from the full-text index. The fts\_\*\_deleted\_cache is the in-memory version of the fts\_\*\_deleted table.

• fts\_\*\_being\_deleted and fts\_\*\_being\_deleted\_cache

Contain the document IDs (DOC\_ID) for documents that are deleted and whose data is currently in the process of being removed from the full-text index. The fts\_\*\_being\_deleted\_cache table is the in-memory version of the fts\_\*\_being\_deleted table.

• fts\_\*\_config

Stores information about the internal state of the full-text index. Most importantly, it stores the FTS\_SYNCED\_DOC\_ID, which identifies documents that have been parsed and flushed to disk. In case of crash recovery, FTS\_SYNCED\_DOC\_ID values are used to identify documents that have not been flushed to disk so that the documents can be re-parsed and added back to the full-text index cache. To view the data in this table, query the Information Schema INNODB\_FT\_CONFIG table.

**InnoDB** **Full-Text** **Index** **Cache**

When a document is inserted, it is tokenized, and the individual words and associated data are inserted into the full-text index. This process, even for small documents, can result in numerous small insertions into the auxiliary index tables, making concurrent access to these tables a point of contention. To avoid this problem, InnoDB uses a full-text index cache to temporarily cache index table insertions for recently inserted rows. This in-memory cache structure holds insertions until the cache is full and then batch flushes them to disk (to the auxiliary index tables). You can query the Information Schema INNODB\_FT\_INDEX\_CACHE table to view tokenized data for recently inserted rows.

The caching and batch flushing behavior avoids frequent updates to auxiliary index tables, which could result in concurrent access issues during busy insert and update times. The batching technique also avoids multiple insertions for the same word, and minimizes duplicate entries. Instead of flushing each word individually, insertions for the same word are merged and flushed to disk as a single entry, improving insertion efficiency while keeping auxiliary index tables as small as possible.

The innodb\_ft\_cache\_size variable is used to configure the full-text index cache size (on a per-table basis), which affects how often the full-text index cache is flushed. You can also define a global full-text index cache size limit for all tables in a given instance using the innodb\_ft\_total\_cache\_size variable.

The full-text index cache stores the same information as auxiliary index tables. However, the full-text index cache only caches tokenized data for recently inserted rows. The data that is already flushed to disk (to the auxiliary index tables) is not brought back into the full-text index cache when queried. The data in auxiliary index tables is queried directly, and results from the auxiliary index tables are merged with results from the full-text index cache before being returned.

**InnoDB** **Full-Text** **Index** **DOC\_ID** **and** **FTS\_DOC\_ID** **Column**

InnoDB uses a unique document identifier referred to as the DOC\_ID to map words in the full-text index to document records where the word appears. The mapping requires an FTS\_DOC\_ID column on the indexed table. If an FTS\_DOC\_ID column is not defined, InnoDB automatically adds a hidden FTS\_DOC\_ID column when the full-text index is created. The following example demonstrates this behavior.

The following table definition does not include an FTS\_DOC\_ID column:

mysql> **CREATE** **TABLE** **opening\_lines** **(**

**id** **INT** **UNSIGNED** **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,**

**opening\_line** **TEXT(500),**

**author** **VARCHAR(200),**

**title** **VARCHAR(200)**

**)** **ENGINE=InnoDB;**

When you create a full-text index on the table using CREATE FULLTEXT INDEX syntax, a warning is returned which reports that InnoDB is rebuilding the table to add the FTS\_DOC\_ID column.

mysql> **CREATE** **FULLTEXT** **INDEX** **idx** **ON** **opening\_lines(opening\_line);**

Query OK, 0 rows affected, 1 warning (0.19 sec)

Records: 0 Duplicates: 0 Warnings: [1](#_bookmark10)

mysql> **SHOW** **WARNINGS;**

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

| Level | Code | Message |

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

| Warning | 124 | InnoDB rebuilding table to add column FTS\_DOC\_ID |

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

The same warning is returned when using ALTER TABLE to add a full-text index to a table that does not have an FTS\_DOC\_ID column. If you create a full-text index at CREATE TABLE time and do not specify an FTS\_DOC\_ID column, InnoDB adds a hidden FTS\_DOC\_ID column, without warning.

Defining an FTS\_DOC\_ID column at CREATE TABLE time is less expensive than creating a full-text index on a table that is already loaded with data. If an FTS\_DOC\_ID column is defined on a table prior



to loading data, the table and its indexes do not have to be rebuilt to add the new column. If you are not concerned with CREATE FULLTEXT INDEX performance, leave out the FTS\_DOC\_ID column to have InnoDB create it for you. InnoDB creates a hidden FTS\_DOC\_ID column along with a unique index (FTS\_DOC\_ID\_INDEX) on the FTS\_DOC\_ID column. If you want to create your own FTS\_DOC\_ID column, the column must be defined as BIGINT UNSIGNED NOT NULL and named FTS\_DOC\_ID (all uppercase), as in the following example:

**Note**

The FTS\_DOC\_ID column does not need to be defined as an AUTO\_INCREMENT column, but doing so could make loading data easier.

|  |  |
| --- | --- |
| mysql> **CREATE** **TABLE** **opening\_lines**  **FTS\_DOC\_ID** **BIGINT** **UNSIGNED**  **opening\_line** **TEXT(500),**  **author** **VARCHAR(200),**  **title** **VARCHAR(200)**  **)** **ENGINE=InnoDB;** | **(**  **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,** |

If you choose to define the FTS\_DOC\_ID column yourself, you are responsible for managing the column to avoid empty or duplicate values. FTS\_DOC\_ID values cannot be reused, which means FTS\_DOC\_ID values must be ever increasing.

Optionally, you can create the required unique FTS\_DOC\_ID\_INDEX (all uppercase) on the FTS\_DOC\_ID column.

mysql> **CREATE** **UNIQUE** **INDEX** **FTS\_DOC\_ID\_INDEX** **on** **opening\_lines(FTS\_DOC\_ID);**

If you do not create the FTS\_DOC\_ID\_INDEX, InnoDB creates it automatically.

**Note**

FTS\_DOC\_ID\_INDEX cannot be defined as a descending index because the InnoDB SQL parser does not use descending indexes.

The permitted gap between the largest used FTS\_DOC\_ID value and new FTS\_DOC\_ID value is

65535.

To avoid rebuilding the table, the FTS\_DOC\_ID column is retained when dropping a full-text index.

**InnoDB** **Full-Text** **Index** **Deletion** **Handling**

Deleting a record that has a full-text index column could result in numerous small deletions in the auxiliary index tables, making concurrent access to these tables a point of contention. To avoid this problem, the DOC\_ID of a deleted document is logged in a special FTS\_\*\_DELETED table whenever a record is deleted from an indexed table, and the indexed record remains in the full-text index. Before returning query results, information in the FTS\_\*\_DELETED table is used to filter out deleted DOC\_IDs. The benefit of this design is that deletions are fast and inexpensive. The drawback is that the size of the index is not immediately reduced after deleting records. To remove full-text index entries for deleted records, run OPTIMIZE TABLE on the indexed table with innodb\_optimize\_fulltext\_only=ON to rebuild the full-text index. For more information, see Optimizing InnoDB Full-Text Indexes.

**InnoDB** **Full-Text** **Index** **Transaction** **Handling**

InnoDB full-text indexes have special transaction handling characteristics due its caching and batch processing behavior. Specifically, updates and insertions on a full-text index are processed at transaction commit time, which means that a full-text search can only see committed data. The following example demonstrates this behavior. The full-text search only returns a result after the inserted lines are committed.

mysql> **CREATE** **TABLE** **opening\_lines** **(**

**id** **INT** **UNSIGNED** **AUTO\_INCREMENT** **NOT** **NULL** **PRIMARY** **KEY,**

**('It** **was** **a** **pleasure** **to** **burn.','Ray** **Bradbury','Fahrenheit** **451');**

mysql> **SELECT** **COUNT(\*)** **FROM** **opening\_lines** **WHERE** **MATCH(opening\_line)** **AGAINST('Ishmael');**

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

| COUNT(\*) |

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

| 0 |

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

mysql> **COMMIT;**

mysql> **SELECT** **COUNT(\*)** **FROM** **opening\_lines** **WHERE** **MATCH(opening\_line)** **AGAINST('Ishmael');**

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

| COUNT(\*) |

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

| 1 |

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

**opening\_line** **TEXT(500),**

**author** **VARCHAR(200),**

**title** **VARCHAR(200),**

**FULLTEXT** **idx** **(opening\_line)**

**)** **ENGINE=InnoDB;**

mysql> **BEGIN;**

mysql> **INSERT** **INTO** **opening\_lines(opening\_line,author,title)** **VALUES**

**('Call** **me** **Ishmael.','Herman** **Melville','Moby-Dick'),**

**('A** **screaming** **comes** **across** **the** **sky.','Thomas** **Pynchon','Gravity\'s** **Rainbow'),**

**('I** **am** **an** **invisible** **man.','Ralph** **Ellison','Invisible** **Man'),**

**('Where** **now?** **Who** **now?** **When** **now?','Samuel** **Beckett','The** **Unnamable'),**

**('It** **was** **love** **at** **first** **sight.','Joseph** **Heller','Catch-22'),**

**('All** **this** **happened,** **more** **or** **less.','Kurt** **Vonnegut','Slaughterhouse-Five'),**

**('Mrs.** **Dalloway** **said** **she** **would** **buy** **the** **flowers** **herself.','Virginia** **Woolf','Mrs.** **Dalloway'),**

**Monitoring** **InnoDB** **Full-Text** **Indexes**

You can monitor and examine the special text-processing aspects of InnoDB full-text indexes by querying the following INFORMATION\_SCHEMA tables:

• INNODB\_FT\_CONFIG

• INNODB\_FT\_INDEX\_TABLE

• INNODB\_FT\_INDEX\_CACHE

• INNODB\_FT\_DEFAULT\_STOPWORD

• INNODB\_FT\_DELETED

• INNODB\_FT\_BEING\_DELETED

You can also view basic information for full-text indexes and tables by querying INNODB\_INDEXES and

INNODB\_TABLES.

For more information, see Section 15.15.4, “InnoDB INFORMATION\_SCHEMA FULLTEXT Index

Tables” .

**15.6.3** **Tablespaces**

This section covers topics related to InnoDB tablespaces.

**15.6.3.1** **The** **System** **Tablespace**

The system tablespace is the storage area for the change buffer. It may also contain table and index data if tables are created in the system tablespace rather than file-per-table or general tablespaces. In previous MySQL versions, the system tablespace contained the InnoDB data dictionary. In MySQL

8.0, InnoDB stores metadata in the MySQL data dictionary. See Chapter 14, *MySQL* *Data* *Dictionary*. In previous MySQL releases, the system tablespace also contained the doublewrite buffer storage area. This storage area resides in separate doublewrite files as of MySQL 8.0.20. See [Section 15.6.4,](#_bookmark11) [“Doublewrite Buffer”](#_bookmark11) .

The system tablespace can have one or more data files. By default, a single system tablespace data file, named ibdata1, is created in the data directory. The size and number of system tablespace data files is defined by the innodb\_data\_file\_path startup option. For configuration information, see System Tablespace Data File Configuration.

Additional information about the system tablespace is provided under the following topics in the section:

• [Resizing the System Tablespace](#_bookmark12)

• [Using Raw Disk Partitions for the System Tablespace](#_bookmark13)

**Resizing** **the** **System** **Tablespace**

This section describes how to increase or decrease the size of the system tablespace.

**Increasing** **the** **Size** **of** **the** **System** **Tablespace**

The easiest way to increase the size of the system tablespace is to configure it to be auto-extending. To do so, specify the autoextend attribute for the last data file in the innodb\_data\_file\_path setting, and restart the server. For example:

innodb\_data\_file\_path=ibdata1:10M:autoextend

When the autoextend attribute is specified, the data file automatically increases in size by 8MB increments as space is required. The innodb\_autoextend\_increment variable controls the increment size.

You can also increase system tablespace size by adding another data file. To do so:

1. Stop the MySQL server.

2. If the last data file in the innodb\_data\_file\_path setting is defined with the autoextend attribute, remove it, and modify the size attribute to reflect the current data file size. To determine the appropriate data file size to specify, check your file system for the file size, and round that value down to the closest MB value, where a MB is equal to 1024 x 1024 bytes.

3. Append a new data file to the innodb\_data\_file\_path setting, optionally specifying the

autoextend attribute. The autoextend attribute can be specified only for the last data file in the innodb\_data\_file\_path setting.

4. Start the MySQL server.

For example, this tablespace has one auto-extending data file:

innodb\_data\_home\_dir =

innodb\_data\_file\_path = /ibdata/ibdata1:10M:autoextend

Suppose that the data file has grown to 988MB over time. This is the innodb\_data\_file\_path setting after modifying the size attribute to reflect the current data file size, and after specifying a new 50MB auto-extending data file:

innodb\_data\_home\_dir =

innodb\_data\_file\_path = /ibdata/ibdata1:988M;/disk2/ibdata2:50M:autoextend

When adding a new data file, do not specify an existing file name. InnoDB creates and initializes the new data file when you start the server.



**Note**

You cannot increase the size of an existing system tablespace

data file by changing its size attribute. For example, changing the innodb\_data\_file\_path setting from ibdata1:10M:autoextend to ibdata1:12M:autoextend produces the following error when starting the server:

[ERROR] [MY-012263] [InnoDB] The Auto-extending innodb\_system

data file ' ./ibdata1' is of a different size 640 pages (rounded down to MB) than

specified in the .cnf file: initial 768 pages, max 0 (relevant if non-zero) pages!

The error indicates that the existing data file size (expressed in InnoDB pages) is different from the data file size specified in the configuration file. If you encounter this error, restore the previous innodb\_data\_file\_path setting, and refer to the system tablespace resizing instructions.

**Decreasing** **the** **Size** **of** **the** **InnoDB** **System** **Tablespace**

Decreasing the size of an existing system tablespace is not supported. The only option to achieve a smaller system tablespace is to restore your data from a backup to a new MySQL instance created with the desired system tablespace size configuration.

For information about creating backups, see Section 15.18.1, “InnoDB Backup” .

For information about configuring data files for a new system tablespace. See System Tablespace Data File Configuration.

To avoid a large system tablespace, consider using file-per-table tablespaces or general tablespaces for your data. File-per-table tablespaces are the default tablespace type and are used implicitly when creating an InnoDB table. Unlike the system tablespace, file-per-table tablespaces return disk space to the operating system when they are truncated or dropped. For more information, see [Section 15.6.3.2,](#_bookmark14) [“File-Per-Table Tablespaces”](#_bookmark14) . General tablespaces are multi-table tablespaces that can also be used as an alternative to the system tablespace. See [Section 15.6.3.3, “General Tablespaces”](#_bookmark15) .

**Using** **Raw** **Disk** **Partitions** **for** **the** **System** **Tablespace**

Raw disk partitions can be used as system tablespace data files. This technique enables nonbuffered I/ O on Windows and some Linux and Unix systems without file system overhead. Perform tests with and without raw partitions to verify whether they improve performance on your system.

When using a raw disk partition, ensure that the user ID that runs the MySQL server has read and write privileges for that partition. For example, if running the server as the mysql user, the partition must be readable and writeable by mysql. If running the server with the --memlock option, the server must be run as root, so the partition must be readable and writeable by root.

The procedures described below involve option file modification. For additional information, see Section 4.2.2.2, “Using Option Files” .

**Allocating** **a** **Raw** **Disk** **Partition** **on** **Linux** **and** **Unix** **Systems**

1. When creating a new data file, specify the keyword newraw immediately after the data file size for the innodb\_data\_file\_path option. The partition must be at least as large as the size that you specify. Note that 1MB in InnoDB is 1024 × 1024 bytes, whereas 1MB in disk specifications usually means 1,000,000 bytes.

[mysqld]

innodb\_data\_home\_dir=

innodb\_data\_file\_path=/dev/hdd1:3Gnewraw;/dev/hdd2:2Gnewraw

2. Restart the server. InnoDB notices the newraw keyword and initializes the new partition. However, do not create or change any InnoDB tables yet. Otherwise, when you next restart the server,

InnoDB reinitializes the partition and your changes are lost. (As a safety measure InnoDB prevents users from modifying data when any partition with newraw is specified.)

3. After InnoDB has initialized the new partition, stop the server, change newraw in the data file specification to raw:

[mysqld]

innodb\_data\_home\_dir=

innodb\_data\_file\_path=/dev/hdd1:3Graw;/dev/hdd2:2Graw

4. Restart the server. InnoDB now permits changes to be made.

**Allocating** **a** **Raw** **Disk** **Partition** **on** **Windows**

On Windows systems, the same steps and accompanying guidelines described for Linux and Unix systems apply except that the innodb\_data\_file\_path setting differs slightly on Windows.

1. When creating a new data file, specify the keyword newraw immediately after the data file size for the innodb\_data\_file\_path option:

[mysqld]

innodb\_data\_home\_dir=

innodb\_data\_file\_path=//./D::10Gnewraw

The //./ corresponds to the Windows syntax of \\.\ for accessing physical drives. In the example above, D: is the drive letter of the partition.

2. Restart the server. InnoDB notices the newraw keyword and initializes the new partition.

3. After InnoDB has initialized the new partition, stop the server, change newraw in the data file specification to raw:

[mysqld]

innodb\_data\_home\_dir=

innodb\_data\_file\_path=//./D::10Graw

4. Restart the server. InnoDB now permits changes to be made.

**15.6.3.2** **File-Per-Table** **Tablespaces**

A file-per-table tablespace contains data and indexes for a single InnoDB table, and is stored on the file system in a single data file.

File-per-table tablespace characteristics are described under the following topics in this section:

• [File-Per-Table Tablespace Configuration](#_bookmark16)

• [File-Per-Table Tablespace Data Files](#_bookmark17)

• [File-Per-Table Tablespace Advantages](#_bookmark18)

• [File-Per-Table Tablespace Disadvantages](#_bookmark19)

**File-Per-Table** **Tablespace** **Configuration**

InnoDB creates tables in file-per-table tablespaces by default. This behavior is controlled by the innodb\_file\_per\_table variable. Disabling innodb\_file\_per\_table causes InnoDB to create tables in the system tablespace.

An innodb\_file\_per\_table setting can be specified in an option file or configured at runtime using a SET GLOBAL statement. Changing the setting at runtime requires privileges sufficient to set global system variables. See Section 5.1.9.1, “System Variable Privileges” .

Option file:

[mysqld]

innodb\_file\_per\_table=ON

Using SET GLOBAL at runtime:

mysql> SET GLOBAL innodb\_file\_per\_table=ON;

**File-Per-Table** **Tablespace** **Data** **Files**

A file-per-table tablespace is created in an .ibd data file in a schema directory under the MySQL data directory. The .ibd file is named for the table (*table\_name*.ibd). For example, the data file for table test.t1 is created in the test directory under the MySQL data directory:

mysql> USE test;

mysql> CREATE TABLE t1 (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(100)

) ENGINE = InnoDB;

$> cd /*path*/*to*/*mysql*/data/test

$> ls

t1.ibd

You can use the DATA DIRECTORY clause of the CREATE TABLE statement to implicitly create a file- per-table tablespace data file outside of the data directory. For more information, see Section 15.6.1.2, “Creating Tables Externally” .

**File-Per-Table** **Tablespace** **Advantages**

File-per-table tablespaces have the following advantages over shared tablespaces such as the system tablespace or general tablespaces.

• Disk space is returned to the operating system after truncating or dropping a table created in a file- per-table tablespace. Truncating or dropping a table stored in a shared tablespace creates free space within the shared tablespace data file, which can only be used for InnoDB data. In other words, a shared tablespace data file does not shrink in size after a table is truncated or dropped.

• A table-copying ALTER TABLE operation on a table that resides in a shared tablespace can increase the amount of disk space occupied by the tablespace. Such operations may require as much additional space as the data in the table plus indexes. This space is not released back to the operating system as it is for file-per-table tablespaces.

• TRUNCATE TABLE performance is better when executed on tables that reside in file-per-table tablespaces.

• File-per-table tablespace data files can be created on separate storage devices for I/O optimization, space management, or backup purposes. See Section 15.6.1.2, “Creating Tables Externally” .

• You can import a table that resides in file-per-table tablespace from another MySQL instance. See Section 15.6.1.3, “Importing InnoDB Tables” .

• Tables created in file-per-table tablespaces support features associated with DYNAMIC and COMPRESSED row formats, which are not supported by the system tablespace. See Section 15.10, “InnoDB Row Formats” .

• Tables stored in individual tablespace data files can save time and improve chances for a successful recovery when data corruption occurs, when backups or binary logs are unavailable, or when the

MySQL server instance cannot be restarted.

• Tables created in file-per-table tablespaces can be backed up or restored quickly using MySQL Enterprise Backup, without interrupting the use of other InnoDB tables. This is beneficial for tables on varying backup schedules or that require backup less frequently. See [Making a Partial Backup](https://dev.mysql.com/doc/mysql-enterprise-backup/8.0/en/partial.html) for details.

• File-per-table tablespaces permit monitoring table size on the file system by monitoring the size of the tablespace data file.

• Common Linux file systems do not permit concurrent writes to a single file such as a shared tablespace data file when innodb\_flush\_method is set to O\_DIRECT. As a result, there are possible performance improvements when using file-per-table tablespaces in conjunction with this setting.

• Tables in a shared tablespace are limited in size by the 64TB tablespace size limit. By comparison, each file-per-table tablespace has a 64TB size limit, which provides plenty of room for individual

tables to grow in size.

**File-Per-Table** **Tablespace** **Disadvantages**

File-per-table tablespaces have the following disadvantages compared to shared tablespaces such as the system tablespace or general tablespaces.

• With file-per-table tablespaces, each table may have unused space that can only be utilized by rows of the same table, which can lead to wasted space if not properly managed.

• fsync operations are performed on multiple file-per-table data files instead of a single shared tablespace data file. Because fsync operations are per file, write operations for multiple tables cannot be combined, which can result in a higher total number of fsync operations.

• mysqld must keep an open file handle for each file-per-table tablespace, which may impact performance if you have numerous tables in file-per-table tablespaces.

• More file descriptors are required when each table has its own data file.

• There is potential for more fragmentation, which can impede DROP TABLE and table scan performance. However, if fragmentation is managed, file-per-table tablespaces can improve

performance for these operations.

• The buffer pool is scanned when dropping a table that resides in a file-per-table tablespace, which can take several seconds for large buffer pools. The scan is performed with a broad internal lock, which may delay other operations.

• The innodb\_autoextend\_increment variable, which defines the increment size for extending the size of an auto-extending shared tablespace file when it becomes full, does not apply to file-per-table tablespace files, which are auto-extending regardless of the innodb\_autoextend\_increment setting. Initial file-per-table tablespace extensions are by small amounts, after which extensions occur in increments of 4MB.

**15.6.3.3** **General** **Tablespaces**

A general tablespace is a shared InnoDB tablespace that is created using CREATE TABLESPACE syntax. General tablespace capabilities and features are described under the following topics in this section:

• [General Tablespace Capabilities](#_bookmark20)

• [Creating a General Tablespace](#_bookmark21)

• [Adding Tables to a General Tablespace](#_bookmark22)

• [General Tablespace Row Format Support](#_bookmark23)

• [Moving Tables Between Tablespaces Using ALTER TABLE](#_bookmark24)

• [Renaming a General Tablespace](#_bookmark25)

• [Dropping a General Tablespace](#_bookmark26)

• [General Tablespace Limitations](#_bookmark27)

**General** **Tablespace** **Capabilities**

General tablespaces provide the following capabilities:

• Similar to the system tablespace, general tablespaces are shared tablespaces capable of storing data for multiple tables.

• General tablespaces have a potential memory advantage over [file-per-table tablespaces](#_bookmark14). The server keeps tablespace metadata in memory for the lifetime of a tablespace. Multiple tables in fewer general tablespaces consume less memory for tablespace metadata than the same number of tables in separate file-per-table tablespaces.

• General tablespace data files can be placed in a directory relative to or independent of the MySQL data directory, which provides you with many of the data file and storage management capabilities of [file-per-table tablespaces](#_bookmark14). As with file-per-table tablespaces, the ability to place data files outside of the MySQL data directory allows you to manage performance of critical tables separately, setup RAID or DRBD for specific tables, or bind tables to particular disks, for example.

• General tablespaces support all table row formats and associated features.

• The TABLESPACE option can be used with CREATE TABLE to create tables in a general tablespaces, file-per-table tablespace, or in the system tablespace.

• The TABLESPACE option can be used with ALTER TABLE to move tables between general tablespaces, file-per-table tablespaces, and the system tablespace.

**Creating** **a** **General** **Tablespace**

General tablespaces are created using CREATE TABLESPACE syntax.

CREATE TABLESPACE *tablespace\_name*

[ADD DATAFILE '*file\_name* ']

[FILE\_BLOCK\_SIZE = *value*]

[ENGINE [=] *engine\_name*]

A general tablespace can be created in the data directory or outside of it. To avoid conflicts with implicitly created file-per-table tablespaces, creating a general tablespace in a subdirectory under the data directory is not supported. When creating a general tablespace outside of the data directory, the directory must exist and must be known to InnoDB prior to creating the tablespace. To make an unknown directory known to InnoDB, add the directory to the innodb\_directories argument value.

innodb\_directories is a read-only startup option. Configuring it requires restarting the server.

Examples:

Creating a general tablespace in the data directory:

|  |  |
| --- | --- |
| mysql> **CREATE** **TABLESPACE** | **`ts1`** **ADD** **DATAFILE** **'ts1.ibd'** **Engine=InnoDB;** |
| or |  |
| mysql> **CREATE** **TABLESPACE** | **`ts1`** **Engine=InnoDB;** |

The ADD DATAFILE clause is optional as of MySQL 8.0.14 and required before that. If the ADD DATAFILE clause is not specified when creating a tablespace, a tablespace data file with a unique file name is created implicitly. The unique file name is a 128 bit UUID formatted into five groups of hexadecimal numbers separated by dashes (*aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee*). General tablespace data files include an .ibd file extension. In a replication environment, the data file name created on the source is not the same as the data file name created on the replica.

Creating a general tablespace in a directory outside of the data directory:

mysql> **CREATE** **TABLESPACE** **`ts1`** **ADD** **DATAFILE** **'/my/tablespace/directory/ts1.ibd'** **Engine=InnoDB;**



You can specify a path that is relative to the data directory as long as the tablespace directory is not under the data directory. In this example, the my\_tablespace directory is at the same level as the data directory:

mysql> **CREATE** **TABLESPACE** **`ts1`** **ADD** **DATAFILE** **'../my\_tablespace/ts1.ibd'** **Engine=InnoDB;**

**Note**

The ENGINE = InnoDB clause must be defined as part of the CREATE TABLESPACE statement, or InnoDB must be defined as the default storage engine (default\_storage\_engine=InnoDB).

**Adding** **Tables** **to** **a** **General** **Tablespace**

After creating a general tablespace, CREATE TABLE *tbl\_name* ... TABLESPACE [=] *tablespace\_name* or ALTER TABLE *tbl\_name* TABLESPACE [=] *tablespace\_name* statements can be used to add tables to the tablespace, as shown in the following examples:

CREATE TABLE:

mysql> **CREATE** **TABLE** **t1** **(c1** **INT** **PRIMARY** **KEY)** **TABLESPACE** **ts1;**

ALTER TABLE:

mysql> **ALTER** **TABLE** **t2** **TABLESPACE** **ts1;**

**Note**

Support for adding table partitions to shared tablespaces was deprecated in MySQL 5.7.24 and removed in MySQL 8.0.13. Shared tablespaces include the InnoDB system tablespace and general tablespaces.

For detailed syntax information, see CREATE TABLE and ALTER TABLE.

**General** **Tablespace** **Row** **Format** **Support**

General tablespaces support all table row formats (REDUNDANT, COMPACT, DYNAMIC, COMPRESSED) with the caveat that compressed and uncompressed tables cannot coexist in the same general tablespace due to different physical page sizes.

For a general tablespace to contain compressed tables (ROW\_FORMAT=COMPRESSED), the FILE\_BLOCK\_SIZE option must be specified, and the FILE\_BLOCK\_SIZE value must be a valid compressed page size in relation to the innodb\_page\_size value. Also, the physical page size of the compressed table (KEY\_BLOCK\_SIZE) must be equal to FILE\_BLOCK\_SIZE/1024. For example, if innodb\_page\_size=16KB and FILE\_BLOCK\_SIZE=8K, the KEY\_BLOCK\_SIZE of the table must be

8.

The following table shows permitted innodb\_page\_size, FILE\_BLOCK\_SIZE, and KEY\_BLOCK\_SIZE combinations. FILE\_BLOCK\_SIZE values may also be specified in bytes. To determine a valid KEY\_BLOCK\_SIZE value for a given FILE\_BLOCK\_SIZE, divide the FILE\_BLOCK\_SIZE value by 1024. Table compression is not support for 32K and 64K InnoDB page sizes. For more information about KEY\_BLOCK\_SIZE, see CREATE TABLE, and Section 15.9.1.2, “Creating Compressed Tables” .

**Table** **15.3** **Permitted** **Page** **Size,** **FILE\_BLOCK\_SIZE,** **and** **KEY\_BLOCK\_SIZE** **Combinations** **for** **Compressed** **Tables**

|  |  |  |
| --- | --- | --- |
| **InnoDB** **Page** **Size**  **(innodb\_page\_size)** | **Permitted** **FILE\_BLOCK\_SIZE** **Value** | **Permitted** **KEY\_BLOCK\_SIZE** **Value** |
| 64KB | 64K (65536) | Compression is not supported |
| 32KB | 32K (32768) | Compression is not supported |



|  |  |  |
| --- | --- | --- |
| **InnoDB** **Page** **Size**  **(innodb\_page\_size)** | **Permitted** **FILE\_BLOCK\_SIZE** **Value** | **Permitted** **KEY\_BLOCK\_SIZE** **Value** |
| 16KB | 16K (16384) | None. If innodb\_page\_size is equal to FILE\_BLOCK\_SIZE, the tablespace cannot contain a compressed table. |
| 16KB | 8K (8192) | 8 |
| 16KB | 4K (4096) | 4 |
| 16KB | 2K (2048) | 2 |
| 16KB | 1K (1024) | 1 |
| 8KB | 8K (8192) | None. If innodb\_page\_size is equal to FILE\_BLOCK\_SIZE, the tablespace cannot contain a compressed table. |
| 8KB | 4K (4096) | 4 |
| 8KB | 2K (2048) | 2 |
| 8KB | 1K (1024) | 1 |
| 4KB | 4K (4096) | None. If innodb\_page\_size is equal to FILE\_BLOCK\_SIZE, the tablespace cannot contain a compressed table. |
| 4KB | 2K (2048) | 2 |
| 4KB | 1K (1024) | 1 |

This example demonstrates creating a general tablespace and adding a compressed table. The example assumes a default innodb\_page\_size of 16KB. The FILE\_BLOCK\_SIZE of 8192 requires that the compressed table have a KEY\_BLOCK\_SIZE of 8.

mysql> **CREATE** **TABLESPACE** **`ts2`** **ADD** **DATAFILE** **'ts2.ibd'** **FILE\_BLOCK\_SIZE** **=** **8192** **Engine=InnoDB;**

mysql> **CREATE** **TABLE** **t4** **(c1** **INT** **PRIMARY** **KEY)** **TABLESPACE** **ts2** **ROW\_FORMAT=COMPRESSED** **KEY\_BLOCK\_SIZE=8;**

If you do not specify FILE\_BLOCK\_SIZE when creating a general tablespace, FILE\_BLOCK\_SIZE defaults to innodb\_page\_size. When FILE\_BLOCK\_SIZE is equal to innodb\_page\_size, the tablespace may only contain tables with an uncompressed row format (COMPACT, REDUNDANT, and DYNAMIC row formats).

**Moving** **Tables** **Between** **Tablespaces** **Using** **ALTER** **TABLE**

ALTER TABLE with the TABLESPACE option can be used to move a table to an existing general tablespace, to a new file-per-table tablespace, or to the system tablespace.

**Note**

Support for placing table partitions in shared tablespaces was deprecated in MySQL 5.7.24 and removed MySQL 8.0.13. Shared tablespaces include the InnoDB system tablespace and general tablespaces.

To move a table from a file-per-table tablespace or from the system tablespace to a general tablespace, specify the name of the general tablespace. The general tablespace must exist. See ALTER TABLESPACE for more information.

ALTER TABLE tbl\_name TABLESPACE [=] *tablespace\_name*;

To move a table from a general tablespace or file-per-table tablespace to the system tablespace, specify innodb\_system as the tablespace name.

ALTER TABLE tbl\_name TABLESPACE [=] innodb\_system;

To move a table from the system tablespace or a general tablespace to a file-per-table tablespace,

specify innodb\_file\_per\_table as the tablespace name. ALTER TABLE tbl\_name TABLESPACE [=] innodb\_file\_per\_table;

ALTER TABLE ... TABLESPACE operations cause a full table rebuild, even if the TABLESPACE attribute has not changed from its previous value.

ALTER TABLE TABLESPACE syntax does not support moving a table from a temporary

tablespace to a .pe.ristent tablespace.

The DATA DIRECTORY clause is permitted with CREATE TABLE ... TABLESPACE=innodb\_file\_per\_table but is otherwise not supported for use in combination with the TABLESPACE option. As of MySQL 8.0.21, the directory specified in a DATA DIRECTORY clause must be known to InnoDB. For more information, see Using the DATA DIRECTORY Clause.

Restrictions apply when moving tables from encrypted tablespaces. See Encryption Limitations.

**Renaming** **a** **General** **Tablespace**

Renaming a general tablespace is supported using ALTER TABLESPACE ... RENAME TO syntax.

ALTER TABLESPACE s1 RENAME TO s2;

The CREATE TABLESPACE privilege is required to rename a general tablespace.

RENAME TO operations are implicitly performed in autocommit mode regardless of the autocommit setting.

A RENAME TO operation cannot be performed while LOCK TABLES or FLUSH TABLES WITH READ LOCK is in effect for tables that reside in the tablespace.

Exclusive metadata locks are taken on tables within a general tablespace while the tablespace is renamed, which prevents concurrent DDL. Concurrent DML is supported.

**Dropping** **a** **General** **Tablespace**

The DROP TABLESPACE statement is used to drop an InnoDB general tablespace.

All tables must be dropped from the tablespace prior to a DROP TABLESPACE operation. If the tablespace is not empty, DROP TABLESPACE returns an error.

Use a query similar to the following to identify tables in a general tablespace.

mysql> **SELECT** **a.NAME** **AS** **space\_name,** **b.NAME** **AS** **table\_name** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESPACES** **a,**

**INFORMATION\_SCHEMA.INNODB\_TABLES** **b** **WHERE** **a** **.SPACE=b** **.SPACE** **AND** **a** **.NAME** **LIKE** **'ts1';**

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

| space\_name | table\_name |

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

|

|

|

|

|

|

|

test/t1

test/t2

test/t3

ts1

ts1

ts1

|

|

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

A general InnoDB tablespace is not deleted automatically when the last table in the tablespace is dropped. The tablespace must be dropped explicitly using DROP TABLESPACE *tablespace\_name*.

A general tablespace does not belong to any particular database. A DROP DATABASE operation can drop tables that belong to a general tablespace but it cannot drop the tablespace, even if the DROP DATABASE operation drops all tables that belong to the tablespace.

Similar to the system tablespace, truncating or dropping tables stored in a general tablespace creates free space internally in the general tablespace .ibd data file which can only be used for new InnoDB



data. Space is not released back to the operating system as it is when a file-per-table tablespace is

deleted during a DROP TABLE operation.

This example demonstrates how to drop an InnoDB general tablespace. The general tablespace ts1 is created with a single table. The table must be dropped before dropping the tablespace.

mysql> **CREATE** **TABLESPACE** **`ts1`** **ADD** **DATAFILE** **'ts1.ibd'** **Engine=InnoDB;**

mysql> **CREATE** **TABLE** **t1** **(c1** **INT** **PRIMARY** **KEY)** **TABLESPACE** **ts1** **Engine=InnoDB;**

mysql> **DROP** **TABLE** **t1;**

mysql> **DROP** **TABLESPACE** **ts1;**

**Note**

*tablespace\_name* is a case-sensitive identifier in MySQL.

**General** **Tablespace** **Limitations**

• A generated or existing tablespace cannot be changed to a general tablespace.

• Creation of temporary general tablespaces is not supported.

• General tablespaces do not support temporary tables.

• Similar to the system tablespace, truncating or dropping tables stored in a general tablespace creates free space internally in the general tablespace .ibd data file which can only be used for new InnoDB data. Space is not released back to the operating system as it is for file-per-table tablespaces.

Additionally, a table-copying ALTER TABLE operation on table that resides in a shared tablespace (a general tablespace or the system tablespace) can increase the amount of space used by the tablespace. Such operations require as much additional space as the data in the table plus indexes. The additional space required for the table-copying ALTER TABLE operation is not released back to the operating system as it is for file-per-table tablespaces.

• ALTER TABLE ... DISCARD TABLESPACE and ALTER TABLE ...IMPORT TABLESPACE are not supported for tables that belong to a general tablespace.

• Support for placing table partitions in general tablespaces was deprecated in MySQL 5.7.24 and removed in MySQL 8.0.13.

• The ADD DATAFILE clause is not supported in a replication environment where the source and replica reside on the same host, as it would cause the source and replica to create a tablespace of the same name in the same location, which is not supported. However, if the ADD DATAFILE clause is omitted, the tablespace is created in the data directory with a generated file name that is unique,

which is permitted.

• As of MySQL 8.0.21, general tablespaces cannot be created in the undo tablespace directory (innodb\_undo\_directory) unless that directly is known to InnoDB. Known directories are those defined by the datadir, innodb\_data\_home\_dir, and innodb\_directories variables.

**15.6.3.4** **Undo** **Tablespaces**

Undo tablespaces contain undo logs, which are collections of records containing information about how to undo the latest change by a transaction to a clustered index record.

Undo tablespaces are described under the following topics in this section:

• [Default Undo Tablespaces](#_bookmark29)

• [Undo Tablespace Size](#_bookmark30)

• [Adding Undo Tablespaces](#_bookmark31)

• [Dropping Undo Tablespaces](#_bookmark32)

• [Moving Undo Tablespaces](#_bookmark33)

• [Configuring the Number of Rollback Segments](#_bookmark34)

• [Truncating Undo Tablespaces](#_bookmark35)

• [Undo Tablespace Status Variables](#_bookmark36)

**Default** **Undo** **Tablespaces**

Two default undo tablespaces are created when the MySQL instance is initialized. Default undo tablespaces are created at initialization time to provide a location for rollback segments that must exist before SQL statements can be accepted. A minimum of two undo tablespaces is required to support automated truncation of undo tablespaces. See [Truncating Undo Tablespaces](#_bookmark35).

Default undo tablespaces are created in the location defined by the innodb\_undo\_directory variable. If the innodb\_undo\_directory variable is undefined, default undo tablespaces are created in the data directory. Default undo tablespace data files are named undo\_001 and undo\_002. The corresponding undo tablespace names defined in the data dictionary are innodb\_undo\_001 and innodb\_undo\_002.

As of MySQL 8.0.14, additional undo tablespaces can be created at runtime using SQL. See [Adding](#_bookmark31) [Undo Tablespaces](#_bookmark31).

**Undo** **Tablespace** **Size**

Prior to MySQL 8.0.23, the initial size of an undo tablespace depends on the innodb\_page\_size value. For the default 16KB page size, the initial undo tablespace file size is 10MiB. For 4KB, 8KB, 32KB, and 64KB page sizes, the initial undo tablespace files sizes are 7MiB, 8MiB, 20MiB, and 40MiB, respectively. As of MySQL 8.0.23, the initial undo tablespace size is normally 16MiB. The initial size may differ when a new undo tablespace is created by a truncate operation. In this case, if the file extension size is larger than 16MB, and the previous file extension occurred within the last second, the new undo tablespace is created at a quarter of the size defined by the innodb\_max\_undo\_log\_size variable.

Prior to MySQL 8.0.23, an undo tablespace is extended four extents at a time. From MySQL 8.0.23, an undo tablespace is extended by a minimum of 16MB. To handle aggressive growth, the file extension size is doubled if the previous file extension happened less than 0.1 seconds earlier. Doubling of the extension size can occur multiple times to a maximum of 256MB. If the previous file extension occurred more than 0.1 seconds earlier, the extension size is reduced by half, which can also occur multiple times, to a minimum of 16MB. If the AUTOEXTEND\_SIZE option is defined for an undo tablespace, it is extended by the greater of the AUTOEXTEND\_SIZE setting and the extension size determined by the logic described above. For information about the AUTOEXTEND\_SIZE option, see [Section 15.6.3.9,](#_bookmark37) [“Tablespace AUTOEXTEND\_SIZE Configuration”](#_bookmark37) .

**Adding** **Undo** **Tablespaces**

Because undo logs can become large during long-running transactions, creating additional undo tablespaces can help prevent individual undo tablespaces from becoming too large. As of MySQL 8.0.14, additional undo tablespaces can be created at runtime using CREATE UNDO TABLESPACE syntax.

CREATE UNDO TABLESPACE *tablespace\_name* ADD DATAFILE '*file\_name*.ibu';

The undo tablespace file name must have an .ibu extension. It is not permitted to specify a relative path when defining the undo tablespace file name. A fully qualified path is permitted, but the path must



the innodb\_undo\_tablespaces startup variable. This variable is deprecated and no longer configurable as of MySQL 8.0.14.

Prior to MySQL 8.0.14, increasing the innodb\_undo\_tablespaces setting creates the specified number of undo tablespaces and adds them to the list of active undo tablespaces. Decreasing the innodb\_undo\_tablespaces setting removes undo tablespaces from the list of active undo tablespaces. Undo tablespaces that are removed from the active list remain active until they are no longer used by existing transactions. The innodb\_undo\_tablespaces variable can be configured at runtime using a SET statement or defined in a configuration file.

Prior to MySQL 8.0.14, deactivated undo tablespaces cannot be removed. Manual removal of undo tablespace files is possible after a slow shutdown but is not recommended, as deactivated undo tablespaces may contain active undo

be known to InnoDB. Known paths are those defined by the innodb\_directories variable. Unique undo tablespace file names are recommended to avoid potential file name conflicts when moving or cloning data.

**Note**

In a replication environment, the source and each replica must have its own undo tablespace file directory. Replicating the creation of an undo tablespace file to a common directory would cause a file name conflict.

At startup, directories defined by the innodb\_directories variable are scanned for undo tablespace files. (The scan also traverses subdirectories.) Directories defined by the innodb\_data\_home\_dir, innodb\_undo\_directory, and datadir variables are automatically appended to the innodb\_directories value regardless of whether the innodb\_directories variable is defined explicitly. An undo tablespace can therefore reside in paths defined by any of those variables.

If the undo tablespace file name does not include a path, the undo tablespace is created in the directory defined by the innodb\_undo\_directory variable. If that variable is undefined, the undo tablespace is created in the data directory.

**Note**

The InnoDB recovery process requires that undo tablespace files reside in known directories. Undo tablespace files must be discovered and opened before redo recovery and before other data files are opened to permit uncommitted transactions and data dictionary changes to be rolled back. An undo tablespace not found before recovery cannot be used, which can lead to database inconsistencies. An error message is reported at startup if an undo tablespace known to the data dictionary is not found. The known directory requirement also supports undo tablespace portability. See [Moving Undo](#_bookmark33) [Tablespaces](#_bookmark33).

To create undo tablespaces in a path relative to the data directory, set the innodb\_undo\_directory variable to the relative path, and specify the file name only when creating an undo tablespace.

To view undo tablespace names and paths, query INFORMATION\_SCHEMA.FILES:

SELECT TABLESPACE\_NAME, FILE\_NAME FROM INFORMATION\_SCHEMA.FILES

WHERE FILE\_TYPE LIKE 'UNDO LOG';

A MySQL instance supports up to 127 undo tablespaces including the two default undo tablespaces created when the MySQL instance is initialized.

 **Note**

 Prior to MySQL 8.0.14, additional undo tablespaces are created by configuring





logs for some time after the server is restarted if open transactions were present when shutting down the server. As of MySQL 8.0.14, undo tablespaces can be dropped using DROP UNDO TABALESPACE syntax. See [Dropping Undo](#_bookmark32) [Tablespaces](#_bookmark32).

**Dropping** **Undo** **Tablespaces**

As of MySQL 8.0.14, undo tablespaces created using CREATE UNDO TABLESPACE syntax can be dropped at runtime using DROP UNDO TABALESPACE syntax.

An undo tablespace must be empty before it can be dropped. To empty an undo tablespace, the undo tablespace must first be marked as inactive using ALTER UNDO TABLESPACE syntax so that the tablespace is no longer used for assigning rollback segments to new transactions.

ALTER UNDO TABLESPACE *tablespace\_name* SET INACTIVE;

After an undo tablespace is marked as inactive, transactions currently using rollback segments in the undo tablespace are permitted to finish, as are any transactions started before those transactions are completed. After transactions are completed, the purge system frees the rollback segments in the undo tablespace, and the undo tablespace is truncated to its initial size. (The same process is used when truncating undo tablespaces. See [Truncating Undo Tablespaces](#_bookmark35).) Once the undo tablespace is empty, it can be dropped.

DROP UNDO TABLESPACE *tablespace\_name*;

**Note**

Alternatively, the undo tablespace can be left in an empty state and reactivated later, if needed, by issuing an ALTER UNDO TABLESPACE *tablespace\_name* SET ACTIVE statement.

The state of an undo tablespace can be monitored by querying the Information Schema INNODB\_TABLESPACES table.

SELECT NAME, STATE FROM INFORMATION\_SCHEMA.INNODB\_TABLESPACES

WHERE NAME LIKE '*tablespace\_name*';

An inactive state indicates that rollback segments in an undo tablespace are no longer used by new

transactions. An empty state indicates that an undo tablespace is empty and ready to be dropped, or ready to be made active again using an ALTER UNDO TABLESPACE *tablespace\_name* SET ACTIVE statement. Attempting to drop an undo tablespace that is not empty returns an error.

The default undo tablespaces (innodb\_undo\_001 and innodb\_undo\_002) created when the MySQL instance is initialized cannot be dropped. They can, however, be made inactive using an ALTER UNDO TABLESPACE *tablespace\_name* SET INACTIVE statement. Before a default undo tablespace can be made inactive, there must be an undo tablespace to take its place. A minimum of two active undo tablespaces are required at all times to support automated truncation of undo tablespaces.

**Moving** **Undo** **Tablespaces**

Undo tablespaces created with CREATE UNDO TABLESPACE syntax can be moved while the server is offline to any known directory. Known directories are those defined by the innodb\_directories variable. Directories defined by innodb\_data\_home\_dir, innodb\_undo\_directory, and datadir are automatically appended to the innodb\_directories value regardless of whether the innodb\_directories variable is defined explicitly. Those directories and their subdirectories are scanned at startup for undo tablespaces files. An undo tablespace file moved to any of those directories is discovered at startup and assumed to be the undo tablespace that was moved.

The default undo tablespaces (innodb\_undo\_001 and innodb\_undo\_002) created when the MySQL instance is initialized must reside in the directory defined by the innodb\_undo\_directory

variable. If the innodb\_undo\_directory variable is undefined, default undo tablespaces reside in the data directory. If default undo tablespaces are moved while the server is offline, the server must be started with the innodb\_undo\_directory variable configured to the new directory.

The I/O patterns for undo logs make undo tablespaces good candidates for SSD storage.

**Configuring** **the** **Number** **of** **Rollback** **Segments**

The innodb\_rollback\_segments variable defines the number of rollback segments allocated to each undo tablespace and to the global temporary tablespace. The innodb\_rollback\_segments variable can be configured at startup or while the server is running.

The default setting for innodb\_rollback\_segments is 128, which is also the maximum value. For information about the number of transactions that a rollback segment supports, see [Section 15.6.6,](#_bookmark38) [“Undo Logs”](#_bookmark38) .

**Truncating** **Undo** **Tablespaces**

There are two methods of truncating undo tablespaces, which can be used individually or in combination to manage undo tablespace size. One method is automated, enabled using configuration variables. The other method is manual, performed using SQL statements.

The automated method does not require monitoring undo tablespace size and, once enabled, it performs deactivation, truncation, and reactivation of undo tablespaces without manual intervention. The manual truncation method may be preferable if you want to control when undo tablespaces are taken offline for truncation. For example, you may want to avoid truncating undo tablespaces during peak workload times.

**Automated** **Truncation**

Automated truncation of undo tablespaces requires a minimum of two active undo tablespaces, which ensures that one undo tablespace remains active while the other is taken offline to be truncated. By default, two undo tablespaces are created when the MySQL instance is initialized.

To have undo tablespaces automatically truncated, enable the innodb\_undo\_log\_truncate variable. For example:

mysql> **SET** **GLOBAL** **innodb\_undo\_log\_truncate=ON;**

When the innodb\_undo\_log\_truncate variable is enabled, undo tablespaces that exceed the size limit defined by the innodb\_max\_undo\_log\_size variable are subject to truncation. The innodb\_max\_undo\_log\_size variable is dynamic and has a default value of 1073741824 bytes (1024 MiB).

mysql> **SELECT** **@@innodb\_max\_undo\_log\_size;**

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

| @@innodb\_max\_undo\_log\_size |

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

|  |  |
| --- | --- |
| | | 1073741824 | |

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

When the innodb\_undo\_log\_truncate variable is enabled:

1. Default and user-defined undo tablespaces that exceed the innodb\_max\_undo\_log\_size setting are marked for truncation. Selection of an undo tablespace for truncation is performed in a circular fashion to avoid truncating the same undo tablespace each time.

2. Rollback segments residing in the selected undo tablespace are made inactive so that they are not assigned to new transactions. Existing transactions that are currently using rollback segments are permitted to finish.

3. The purge system empties rollback segments by freeing undo logs that are no longer in use.



4. After all rollback segments in the undo tablespace are freed, the truncate operation runs and truncates the undo tablespace to its initial size.

The size of an undo tablespace after a truncate operation may be larger than the initial size due to immediate use following the completion of the operation.

The innodb\_undo\_directory variable defines the location of default undo tablespace files. If the innodb\_undo\_directory variable is undefined, default undo tablespaces reside in the data directory. The location of all undo tablespace files including user-defined undo tablespaces created using CREATE UNDO TABLESPACE syntax can be determined by querying the Information Schema FILES table:

SELECT TABLESPACE\_NAME, FILE\_NAME FROM INFORMATION\_SCHEMA.FILES WHERE FILE\_TYPE LIKE 'UNDO LOG';

5. Rollback segments are reactivated so that they can be assigned to new transactions.

**Manual** **Truncation**

Manual truncation of undo tablespaces requires a minimum of three active undo tablespaces. Two active undo tablespaces are required at all times to support the possibility that automated truncation is enabled. A minimum of three undo tablespaces satisfies this requirement while permitting an undo tablespace to be taken offline manually.

To manually initiate truncation of an undo tablespace, deactivate the undo tablespace by issuing the following statement:

ALTER UNDO TABLESPACE *tablespace\_name* SET INACTIVE;

After the undo tablespace is marked as inactive, transactions currently using rollback segments in the undo tablespace are permitted to finish, as are any transactions started before those transactions are completed. After transactions are completed, the purge system frees the rollback segments in the undo tablespace, the undo tablespace is truncated to its initial size, and the undo tablespace state changes from inactive to empty.

**Note**

When an ALTER UNDO TABLESPACE *tablespace\_name* SET INACTIVE statement deactivates an undo tablespace, the purge thread looks for that undo tablespace at the next opportunity. Once the undo tablespace is found and marked for truncation, the purge thread returns with increased frequency to quickly empty and truncate the undo tablespace.

To check the state of an undo tablespace, query the Information Schema INNODB\_TABLESPACES table.

SELECT NAME, STATE FROM INFORMATION\_SCHEMA.INNODB\_TABLESPACES

WHERE NAME LIKE '*tablespace\_name*';

Once the undo tablespace is in an empty state, it can be reactivated by issuing the following statement:

ALTER UNDO TABLESPACE *tablespace\_name* SET ACTIVE;

An undo tablespace in an empty state can also be dropped. See [Dropping Undo Tablespaces](#_bookmark32).

**Expediting** **Automated** **Truncation** **of** **Undo** **Tablespaces**

The purge thread is responsible for emptying and truncating undo tablespaces. By default, the purge thread looks for undo tablespaces to truncate once every 128 times that purge is invoked. The frequency with which the purge thread looks for undo tablespaces to truncate is controlled by the innodb\_purge\_rseg\_truncate\_frequency variable, which has a default setting of 128.

mysql> **SELECT** **@@innodb\_purge\_rseg\_truncate\_frequency;**

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

| @@innodb\_purge\_rseg\_truncate\_frequency |

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

| 128 |

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

To increase the frequency, decrease the innodb\_purge\_rseg\_truncate\_frequency setting. For example, to have the purge thread look for undo tablespaces once every 32 times that purge is invoked, set innodb\_purge\_rseg\_truncate\_frequency to 32.

mysql> **SET** **GLOBAL** **innodb\_purge\_rseg\_truncate\_frequency=32;**

**Performance** **Impact** **of** **Truncating** **Undo** **Tablespace** **Files**

When an undo tablespace is truncated, the rollback segments in the undo tablespace are deactivated. The active rollback segments in other undo tablespaces assume responsibility for the entire system load, which may result in a slight performance degradation. The extent to which performance is affected depends on a number of factors:

• Number of undo tablespaces

• Number of undo logs

• Undo tablespace size

• Speed of the I/O subsystem

• Existing long running transactions

• System load

The easiest way to avoid the potential performance impact is to increase the number of undo tablespaces.

**Monitoring** **Undo** **Tablespace** **Truncation**

As of MySQL 8.0.16, undo and purge subsystem counters are provided for monitoring background activities associated with undo log truncation. For counter names and descriptions, query the Information Schema INNODB\_METRICS table.

SELECT NAME, SUBSYSTEM, COMMENT FROM INFORMATION\_SCHEMA.INNODB\_METRICS WHERE NAME LIKE '%truncate%';

For information about enabling counters and querying counter data, see Section 15.15.6, “InnoDB INFORMATION\_SCHEMA Metrics Table” .

**Undo** **Tablespace** **Truncation** **Limit**

As of MySQL 8.0.21, the number of truncate operations on the same undo tablespace between checkpoints is limited to 64. The limit prevents potential issues caused by an excessive number of undo tablespace truncate operations, which can occur if innodb\_max\_undo\_log\_size is set too low on a busy system, for example. If the limit is exceeded, an undo tablespace can still be made inactive, but it is not truncated until after the next checkpoint. the limit was raised from 64 to 50,000 in MySQL

8.0.22.

**Undo** **Tablespace** **Truncation** **Recovery**

An undo tablespace truncate operation creates a temporary undo\_*space\_number*\_trunc.log file in the server log directory. That log directory is defined by innodb\_log\_group\_home\_dir. If a system failure occurs during the truncate operation, the temporary log file permits the startup process to identify undo tablespaces that were being truncated and to continue the operation.

**Undo** **Tablespace** **Status** **Variables**

The following status variables permit tracking the total number of undo tablespaces, implicit ( InnoDB-

created) undo tablespaces, explicit (user-created) undo tablespaces, and the number of active undo tablespaces:

mysql> **SHOW** **STATUS** **LIKE** **'Innodb\_undo\_tablespaces%';**

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

| Variable\_name | Value |

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

|

|

|

|

|

Innodb\_undo\_tablespaces\_total

Innodb\_undo\_tablespaces\_implicit

Innodb\_undo\_tablespaces\_explicit

Innodb\_undo\_tablespaces\_active

|

2

2

0

2

|

|

|

|

|

|

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

For status variable descriptions, see Section 5.1.10, “Server Status Variables” .

**15.6.3.5** **Temporary** **Tablespaces**

InnoDB uses session temporary tablespaces and a global temporary tablespace.

**Session** **Temporary** **Tablespaces**

Session temporary tablespaces store user-created temporary tables and internal temporary tables created by the optimizer when InnoDB is configured as the storage engine for on-disk internal temporary tables. Beginning with MySQL 8.0.16, the storage engine used for on-disk internal temporary tables is InnoDB. (Previously, the storage engine was determined by the value of internal\_tmp\_disk\_storage\_engine.)

Session temporary tablespaces are allocated to a session from a pool of temporary tablespaces on the first request to create an on-disk temporary table. A maximum of two tablespaces is allocated to a session, one for user-created temporary tables and the other for internal temporary tables created by the optimizer. The temporary tablespaces allocated to a session are used for all on-disk temporary tables created by the session. When a session disconnects, its temporary tablespaces are truncated and released back to the pool. A pool of 10 temporary tablespaces is created when the server is started. The size of the pool never shrinks and tablespaces are added to the pool automatically as necessary. The pool of temporary tablespaces is removed on normal shutdown or on an aborted initialization. Session temporary tablespace files are five pages in size when created and have an .ibt file name extension.

A range of 400 thousand space IDs is reserved for session temporary tablespaces. Because the pool of session temporary tablespaces is recreated each time the server is started, space IDs for session temporary tablespaces are not persisted when the server is shut down, and may be reused.

The innodb\_temp\_tablespaces\_dir variable defines the location where session temporary tablespaces are created. The default location is the #innodb\_temp directory in the data directory. Startup is refused if the pool of temporary tablespaces cannot be created.

$> cd *BASED工R*/data/#innodb\_temp

$> ls

temp\_10 .ibt temp\_2 .ibt temp\_4 .ibt temp\_6 .ibt temp\_8 .ibt

temp\_1.ibt temp\_3.ibt temp\_5.ibt temp\_7.ibt temp\_9.ibt

In statement based replication (SBR) mode, temporary tables created on a replica reside in a single session temporary tablespace that is truncated only when the MySQL server is shut down.

The INNODB\_SESSION\_TEMP\_TABLESPACES table provides metadata about session temporary tablespaces.

The Information Schema INNODB\_TEMP\_TABLE\_INFO table provides metadata about user-created temporary tables that are active in an InnoDB instance.

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

The global temporary tablespace (ibtmp1) stores rollback segments for changes made to user- created temporary tables.

The innodb\_temp\_data\_file\_path variable defines the relative path, name, size, and attributes for global temporary tablespace data files. If no value is specified for innodb\_temp\_data\_file\_path, the default behavior is to create a single auto-extending data file named ibtmp1 in the innodb\_data\_home\_dir directory. The initial file size is slightly larger than 12MB.

The global temporary tablespace is removed on normal shutdown or on an aborted initialization, and recreated each time the server is started. The global temporary tablespace receives a dynamically generated space ID when it is created. Startup is refused if the global temporary tablespace cannot be created. The global temporary tablespace is not removed if the server halts unexpectedly. In this case, a database administrator can remove the global temporary tablespace manually or restart the MySQL server. Restarting the MySQL server removes and recreates the global temporary tablespace automatically.

The global temporary tablespace cannot reside on a raw device.

The Information Schema FILES table provides metadata about the global temporary tablespace. Issue a query similar to this one to view global temporary tablespace metadata:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.FILES** **WHERE** **TABLESPACE\_NAME='innodb\_temporary'\G**

By default, the global temporary tablespace data file is autoextending and increases in size as necessary.

To determine if a global temporary tablespace data file is autoextending, check the innodb\_temp\_data\_file\_path setting:

mysql> **SELECT** **@@innodb\_temp\_data\_file\_path;**

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

| @@innodb\_temp\_data\_file\_path |

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

| ibtmp1:12M:autoextend |

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

To check the size of global temporary tablespace data files, examine the Information Schema FILES table using a query similar to this one:

mysql> **SELECT** **FILE\_NAME,** **TABLESPACE\_NAME,** **ENGINE,** **INITIAL\_SIZE,** **TOTAL\_EXTENTS\*EXTENT\_SIZE**

**AS** **TotalSizeBytes,** **DATA\_FREE,** **MAXIMUM\_SIZE** **FROM** **INFORMATION\_SCHEMA.FILES**

**WHERE** **TABLESPACE\_NAME** **=** **'innodb\_temporary'\G**

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

FILE\_NAME: ./ibtmp1

TABLESPACE\_NAME: innodb\_temporary

ENGINE: InnoDB

INITIAL\_SIZE: 12582912

TotalSizeBytes: 12582912

DATA\_FREE: 6291456

MAXIMUM\_SIZE: NULL

TotalSizeBytes shows the current size of the global temporary tablespace data file. For information about other field values, see Section 26.3.15, “The INFORMATION\_SCHEMA FILES Table” .

Alternatively, check the global temporary tablespace data file size on your operating system. The global temporary tablespace data file is located in the directory defined by the innodb\_temp\_data\_file\_path variable.

To reclaim disk space occupied by a global temporary tablespace data file, restart the MySQL server. Restarting the server removes and recreates the global temporary tablespace data file according to the attributes defined by innodb\_temp\_data\_file\_path.

To limit the size of the global temporary tablespace data file, configure

innodb\_temp\_data\_file\_path to specify a maximum file size. For example:

[mysqld]

innodb\_temp\_data\_file\_path=ibtmp1:12M:autoextend:max:500M

Configuring innodb\_temp\_data\_file\_path requires restarting the server.

**15.6.3.6** **Moving** **Tablespace** **Files** **While** **the** **Server** **is** **Offline**

The innodb\_directories variable, which defines directories to scan at startup for tablespace files, supports moving or restoring tablespace files to a new location while the server is offline. During startup, discovered tablespace files are used instead those referenced in the data dictionary, and the data dictionary is updated to reference the relocated files. If duplicate tablespace files are discovered by the scan, startup fails with an error indicating that multiple files were found for the same tablespace

ID.

The directories defined by the innodb\_data\_home\_dir, innodb\_undo\_directory, and datadir variables are automatically appended to the innodb\_directories argument value. These directories are scanned at startup regardless of whether an innodb\_directories setting is specified explicitly. The implicit addition of these directories permits moving system tablespace files, the data directory, or undo tablespace files without configuring the innodb\_directories setting. However, settings must be updated when directories change. For example, after relocating the data directory, you must update the --datadir setting before restarting the server.

The innodb\_directories variable can be specified in a startup command or MySQL option file. Quotes are used around the argument value because a semicolon (;) is interpreted as a special character by some command interpreters. (Unix shells treat it as a command terminator, for example.)

Startup command:

mysqld --innodb-directories="*directory\_path\_1*;*directory\_path\_2*"

MySQL option file:

[mysqld]

innodb\_directories="*directory\_path\_1*;*directory\_path\_2*"

The following procedure is applicable to moving individual file-per-table and general tablespace files, system tablespace files, undo tablespace files, or the data directory. Before moving files or directories, review the usage notes that follow.

1. Stop the server.

2. Move the tablespace files or directories to the desired location.

3. Make the new directory known to InnoDB.

• If moving individual file-per-table or general tablespace files, add unknown directories to the innodb\_directories value.

• The directories defined by the innodb\_data\_home\_dir, innodb\_undo\_directory, and datadir variables are automatically appended to the innodb\_directories argument

value, so you need not specify these.

• A file-per-table tablespace file can only be moved to a directory with same name as the schema. For example, if the actor table belongs to the sakila schema, then the actor.ibd data file can only be moved to a directory named sakila.

• General tablespace files cannot be moved to the data directory or a subdirectory of the data directory.



• If moving system tablespace files, undo tablespaces, or the data directory, update the innodb\_data\_home\_dir, innodb\_undo\_directory, and datadir settings, as necessary.

4. Restart the server.

**Usage** **Notes**

• Wildcard expressions cannot be used in the innodb\_directories argument value.

• The innodb\_directories scan also traverses subdirectories of specified directories. Duplicate directories and subdirectories are discarded from the list of directories to be scanned.

• innodb\_directories supports moving InnoDB tablespace files. Moving files that belong to a storage engine other than InnoDB is not supported. This restriction also applies when moving the entire data directory.

• innodb\_directories supports renaming of tablespace files when moving files to a scanned directory. It also supports moving tablespaces files to other supported operating systems.

• When moving tablespace files to a different operating system, ensure that tablespace file names do not include prohibited characters or characters with a special meaning on the destination system.

• When moving a data directory from a Windows operating system to a Linux operating system, modify the binary log file paths in the binary log index file to use backward slashes instead of forward slashes. By default, the binary log index file has the same base name as the binary log file, with the extension ' .index'. The location of the binary log index file is defined by --log-bin. The default location is the data directory.

• If moving tablespace files to a different operating system introduces cross-platform replication, it is the database administrator's responsibility to ensure proper replication of DDL statements that contain platform-specific directories. Statements that permit specifying directories include CREATE TABLE ... DATA DIRECTORY and CREATE TABLESPACE ... ADD DATAFILE.

• Add the directories of file-per-table and general tablespaces created with an absolute path or in a location outside of the data directory to the innodb\_directories setting. Otherwise, InnoDB is not able to locate the files during recovery. For related information, see Tablespace Discovery During Crash Recovery.

To view tablespace file locations, query the Information Schema FILES table:

mysql> **SELECT** **TABLESPACE\_NAME,** **FILE\_NAME** **FROM** **INFORMATION\_SCHEMA.FILES** **\G**

**15.6.3.7** **Disabling** **Tablespace** **Path** **Validation**

At startup, InnoDB scans directories defined by the innodb\_directories variable for tablespace files. The paths of discovered tablespace files are validated against the paths recorded in the data dictionary. If the paths do not match, the paths in the data dictionary are updated.

The innodb\_validate\_tablespace\_paths variable, introduced in MySQL 8.0.21, permits disabling tablespace path validation. This feature is intended for environments where tablespaces files are not moved. Disabling path validation improves startup time on systems with a large number of tablespace files. If log\_error\_verbosity is set to 3, the following message is printed at startup when tablespace path validation is disabled:

[InnoDB] Skipping InnoDB tablespace path validation.

Manually moved tablespace files will not be detected!

**Warning**

Starting the server with tablespace path validation disabled after moving tablespace files can lead to undefined behavior.

**15.6.3.8** **Optimizing** **Tablespace** **Space** **Allocation** **on** **Linux**

As of MySQL 8.0.22, you can optimize how InnoDB allocates space to file-per-table and

general tablespaces on Linux. By default, when additional space is required, InnoDB allocates pages to the tablespace and physically writes NULLs to those pages. This behavior can affect performance if new pages are allocated frequently. As of MySQL 8.0.22, you can disable innodb\_extend\_and\_initialize on Linux systems to avoid physically writing NULLs to newly allocated tablespace pages. When innodb\_extend\_and\_initialize is disabled, space is allocated to tablespace files using posix\_fallocate() calls, which reserve space without physically writing NULLs.

When pages are allocated using posix\_fallocate() calls, the extension size is small by default and pages are often allocated only a few at a time, which can cause fragmentation and increase random I/O. To avoid this issue, increase the tablespace extension size when enabling posix\_fallocate() calls. Tablespace extension size can be increased up to 4GB using the AUTOEXTEND\_SIZE option. For more information, see [Section 15.6.3.9, “Tablespace](#_bookmark37) [AUTOEXTEND\_SIZE Configuration”](#_bookmark37) .

InnoDB writes a redo log record before allocating a new tablespace page. If a page allocation operation is interrupted, the operation is replayed from the redo log record during recovery. (A page allocation operation replayed from a redo log record physically writes NULLs to the newly allocated page.) A redo log record is written before allocating a page regardless of the innodb\_extend\_and\_initialize setting.

On non-Linux systems and Windows, InnoDB allocates new pages to the tablespace and physically writes NULLs to those pages, which is the default behavior. Attempting to disable innodb\_extend\_and\_initialize on those systems returns the following error:

Changing innodb\_extend\_and\_initialize not supported on this platform.

Falling back to the default.

**15.6.3.9** **Tablespace** **AUTOEXTEND\_SIZE** **Configuration**

By default, when a file-per-table or general tablespace requires additional space, the tablespace is extended incrementally according to the following rules:

• If the tablespace is less than an extent in size, it is extended one page at a time.

• If the tablespace is greater than 1 extent but smaller than 32 extents in size, it is extended one extent at a time.

• If the tablespace is more than 32 extents in size, it is extended four extents at a time. For information about extent size, see Section 15.11.2, “File Space Management” .

From MySQL 8.0.23, the amount by which a file-per-table or general tablespace is extended is configurable by specifying the AUTOEXTEND\_SIZE option. Configuring a larger extension size can help avoid fragmentation and facilitate ingestion of large amounts of data.

To configure the extension size for a file-per-table tablespace, specify the AUTOEXTEND\_SIZE size in a CREATE TABLE or ALTER TABLE statement:

CREATE TABLE t1 (c1 INT) AUTOEXTEND\_SIZE = 4M;

ALTER TABLE t1 AUTOEXTEND\_SIZE = 8M;

To configure the extension size for a general tablespace, specify the AUTOEXTEND\_SIZE size in a CREATE TABLESPACE or ALTER TABLESPACE statement:

CREATE TABLESPACE ts1 AUTOEXTEND\_SIZE = 4M;

ALTER TABLESPACE ts1 AUTOEXTEND\_SIZE = 8M;



**Note**

The AUTOEXTEND\_SIZE option can also be used when creating an undo tablespace, but the extension behavior for undo tablespaces differs. For more information, see [Section 15.6.3.4, “Undo Tablespaces”](#_bookmark28) .

The AUTOEXTEND\_SIZE setting must be a multiple of 4M. Specifying an AUTOEXTEND\_SIZE setting that is not a multiple of 4M returns an error.

The AUTOEXTEND\_SIZE default setting is 0, which causes the tablespace to be extended according to the default behavior described above.

The maximum AUTOEXTEND\_SIZE setting is 64M in MySQL 8.0.23. From MySQL 8.0.24, the maximum setting is 4GB.

The minimum AUTOEXTEND\_SIZE setting depends on the InnoDB page size, as shown in the following table:

|  |  |
| --- | --- |
| **InnoDB** **Page** **Size** | **Minimum** **AUTOEXTEND\_SIZE** |
| 4K | 4M |
| 8K | 4M |
| 16K | 4M |
| 32K | 8M |
| 64K | 16M |

The default InnoDB page size is 16K (16384 bytes). To determine the InnoDB page size for your MySQL instance, query the innodb\_page\_size setting:

mysql> **SELECT** **@@GLOBAL** **.innodb\_page\_size;**

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

| @@GLOBAL .innodb\_page\_size |

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

|  |  |
| --- | --- |
| | | 16384 | |

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

When the AUTOEXTEND\_SIZE setting for a tablespace is altered, the first extension that occurs afterward increases the tablespace size to a multiple of the AUTOEXTEND\_SIZE setting. Subsequent extensions are of the configured size.

When a file-per-table or general tablespace is created with a non-zero AUTOEXTEND\_SIZE setting, the tablespace is initialized at the specified AUTOEXTEND\_SIZE size.

ALTER TABLESPACE cannot be used to configure the AUTOEXTEND\_SIZE of a file-per-table tablespace. ALTER TABLE must be used.

For tables created in file-per-table tablespaces, SHOW CREATE TABLE shows the AUTOEXTEND\_SIZE option only when it is configured to a non-zero value.

To determine the AUTOEXTEND\_SIZE for any InnoDB tablespace, query the Information Schema INNODB\_TABLESPACES table. For example:

mysql> **SELECT** **NAME,** **AUTOEXTEND\_SIZE** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESPACES**

**WHERE** **NAME** **LIKE** **'test/t1';**

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

| NAME | AUTOEXTEND\_SIZE |

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

| test/t1 | 4194304 |

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

mysql> **SELECT** **NAME,** **AUTOEXTEND\_SIZE** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLESPACES**

**WHERE** **NAME** **LIKE** **'ts1';**

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



| NAME | AUTOEXTEND\_SIZE |

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

|  |  |
| --- | --- |
| | ts1 | | 4194304 | |

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

**Note**

An AUTOEXTEND\_SIZE of 0, which is the default setting, means that the tablespace is extended according to the default tablespace extension behavior described above.

**15.6.4** **Doublewrite** **Buffer**

The doublewrite buffer is a storage area where InnoDB writes pages flushed from the buffer pool before writing the pages to their proper positions in the InnoDB data files. If there is an operating system, storage subsystem, or unexpected mysqld process exit in the middle of a page write, InnoDB can find a good copy of the page from the doublewrite buffer during crash recovery.

Although data is written twice, the doublewrite buffer does not require twice as much I/O overhead or twice as many I/O operations. Data is written to the doublewrite buffer in a large sequential chunk, with a single fsync() call to the operating system (except in the case that innodb\_flush\_method is set to O\_DIRECT\_NO\_FSYNC).

Prior to MySQL 8.0.20, the doublewrite buffer storage area is located in the InnoDB system tablespace. As of MySQL 8.0.20, the doublewrite buffer storage area is located in doublewrite files.

The following variables are provided for doublewrite buffer configuration:

• innodb\_doublewrite

The innodb\_doublewrite variable controls whether the doublewrite buffer is enabled. It is enabled by default in most cases. To disable the doublewrite buffer, set innodb\_doublewrite to OFF. Consider disabling the doublewrite buffer if you are more concerned with performance than data integrity, as may be the case when performing benchmarks, for example.

From MySQL 8.0.30, innodb\_doublewrite supports DETECT\_AND\_RECOVER and DETECT\_ONLY settings.

The DETECT\_AND\_RECOVER setting is the same as the ON setting. With this setting, the doublewrite buffer is fully enabled, with database page content written to the doublewrite buffer where it is accessed during recovery to fix incomplete page writes.

With the DETECT\_ONLY setting, only metadata is written to the doublewrite buffer. Database page content is not written to the doublewrite buffer, and recovery does not use the doublewrite buffer to fix incomplete page writes. This lightweight setting is intended for detecting incomplete page writes only.

MySQL 8.0.30 onwards supports dynamic changes to the innodb\_doublewrite setting that enables the doublewrite buffer, between ON, DETECT\_AND\_RECOVER, and DETECT\_ONLY. MySQL does not support dynamic changes between a setting that enables the doublewrite buffer and OFF or vice versa.

If the doublewrite buffer is located on a Fusion-io device that supports atomic writes, the doublewrite buffer is automatically disabled and data file writes are performed using Fusion-io atomic writes instead. However, be aware that the innodb\_doublewrite setting is global. When the doublewrite buffer is disabled, it is disabled for all data files including those that do not reside on Fusion-io hardware. This feature is only supported on Fusion-io hardware and is only enabled for Fusion- io NVMFS on Linux. To take full advantage of this feature, an innodb\_flush\_method setting of O\_DIRECT is recommended.

• innodb\_doublewrite\_dir

The innodb\_doublewrite\_dir variable (introduced in MySQL 8.0.20) defines the directory where InnoDB creates doublewrite files. If no directory is specified, doublewrite files are created in the innodb\_data\_home\_dir directory, which defaults to the data directory if unspecified.

A hash symbol '#' is automatically prefixed to the specified directory name to avoid conflicts with schema names. However, if a '.', '#'. or '/' prefix is specified explicitly in the directory name, the hash symbol '#' is not prefixed to the directory name.

Ideally, the doublewrite directory should be placed on the fastest storage media available.

• innodb\_doublewrite\_files

The innodb\_doublewrite\_files variable defines the number of doublewrite files. By default, two doublewrite files are created for each buffer pool instance: A flush list doublewrite file and an LRU list doublewrite file.

The flush list doublewrite file is for pages flushed from the buffer pool flush list. The default size of a flush list doublewrite file is the InnoDB page size \* doublewrite page bytes.

The LRU list doublewrite file is for pages flushed from the buffer pool LRU list. It also contains slots for single page flushes. The default size of an LRU list doublewrite file is the InnoDB page size \* (doublewrite pages + (512 / the number of buffer pool instances)) where 512 is the total number of slots reserved for single page flushes.

At a minimum, there are two doublewrite files. The maximum number of doublewrite files is two times the number of buffer pool instances. (The number of buffer pool instances is controlled by the innodb\_buffer\_pool\_instances variable.)

Doublewrite file names have the following format: #ib\_*page\_size*\_*file\_number*.dblwr (or .bdblwr with the DETECT\_ONLY setting). For example, the following doublewrite files are created for a MySQL instance with an InnoDB pages size of 16KB and a single buffer pool:

#ib\_16384\_0.dblwr

#ib\_16384\_1.dblwr

The innodb\_doublewrite\_files variable is intended for advanced performance tuning. The default setting should be suitable for most users.

• innodb\_doublewrite\_pages

The innodb\_doublewrite\_pages variable (introduced in MySQL 8.0.20) controls the maximum number of doublewrite pages per thread. If no value is specified, innodb\_doublewrite\_pages is set to the innodb\_write\_io\_threads value. This variable is intended for advanced performance tuning. The default value should be suitable for most users.

• innodb\_doublewrite\_batch\_size

The innodb\_doublewrite\_batch\_size variable (introduced in MySQL 8.0.20) controls the number of doublewrite pages to write in a batch. This variable is intended for advanced performance tuning. The default value should be suitable for most users.

As of MySQL 8.0.23, InnoDB automatically encrypts doublewrite file pages that belong to encrypted tablespaces (see Section 15.13, “InnoDB Data-at-Rest Encryption”). Likewise, doublewrite file pages belonging page-compressed tablespaces are compressed. As a result, doublewrite files can contain different page types including unencrypted and uncompressed pages, encrypted pages, compressed pages, and pages that are both encrypted and compressed.

**15.6.5** **Redo** **Log**

The redo log is a disk-based data structure used during crash recovery to correct data written by incomplete transactions. During normal operations, the redo log encodes requests to change table

data that result from SQL statements or low-level API calls. Modifications that did not finish updating data files before an unexpected shutdown are replayed automatically during initialization and before connections are accepted. For information about the role of the redo log in crash recovery, see Section 15.18.2, “InnoDB Recovery” .

The redo log is physically represented on disk by redo log files. Data that is written to redo log files is encoded in terms of records affected, and this data is collectively referred to as redo. The passage of data through redo log files is represented by an ever-increasing LSN value. Redo log data is appended as data modifications occur, and the oldest data is truncated as the checkpoint progresses.

Information and procedures related to redo logs are described under the following topics in the section:

• [Configuring Redo Log Capacity (MySQL 8.0.30 or Higher)](#_bookmark39)

• [Configuring Redo Log Capacity (Before MySQL 8.0.30)](#_bookmark40)

• [Automatic Redo Log Capacity Configuration](#_bookmark41)

• [Redo Log Archiving](#_bookmark42)

• [Disabling Redo Logging](#_bookmark43)

• [Related Topics](#_bookmark44)

**Configuring** **Redo** **Log** **Capacity** **(MySQL** **8.0.30** **or** **Higher)**

From MySQL 8.0.30, the innodb\_redo\_log\_capacity system variable controls the amount of disk space occupied by redo log files. You can set this variable in an option file at startup or at runtime using a SET GLOBAL statement; for example, the following statement sets the redo log capacity to 8GB:

SET GLOBAL innodb\_redo\_log\_capacity = 8589934592;

When set at runtime, the configuration change occurs immediately but it may take some time for the new limit to be fully implemented. If the redo log files occupy less space than the specified value, dirty pages are flushed from the buffer pool to tablespace data files less aggressively, eventually increasing the disk space occupied by the redo log files. If the redo log files occupy more space than the specified value, dirty pages are flushed more aggressively, eventually decreasing the disk space occupied by redo log files.

The innodb\_redo\_log\_capacity variable supersedes the innodb\_log\_files\_in\_group and innodb\_log\_file\_size variables, which are deprecated. When the innodb\_redo\_log\_capacity setting is defined, the innodb\_log\_files\_in\_group and innodb\_log\_file\_size 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, redo log capacity is set to the innodb\_redo\_log\_capacity default value, which is 104857600 bytes (100MB). The maximum redo log capacity is 128GB.

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. There are two types of redo log files, ordinary and spare. Ordinary redo log files are those being used. Spare redo log files are those waiting to be used. InnoDB tries to maintain 32 redo log files in total, with each file equal in size to 1/32 \* innodb\_redo\_log\_capacity; however, file sizes may differ for a time after modifying the innodb\_redo\_log\_capacity setting.

Redo log files use an #ib\_redo*N* naming convention, where *N* is the redo log file number. Spare redo log files are denoted by a \_tmp suffix. The following example shows the redo log files in an #innodb\_redo directory, where there are 21 active redo log files and 11 spare redo log files, numbered sequentially.

|  |  |  |  |
| --- | --- | --- | --- |
| '#ib\_redo582' | '#ib\_redo590' | '#ib\_redo598' | '#ib\_redo606\_tmp' |

|  |  |  |  |
| --- | --- | --- | --- |
| '#ib\_redo583'  '#ib\_redo584'  '#ib\_redo585'  '#ib\_redo586'  '#ib\_redo587'  '#ib\_redo588'  '#ib\_redo589' | '#ib\_redo591'  '#ib\_redo592'  '#ib\_redo593'  '#ib\_redo594'  '#ib\_redo595'  '#ib\_redo596'  '#ib\_redo597' | '#ib\_redo599'  '#ib\_redo600'  '#ib\_redo601'  '#ib\_redo602'  '#ib\_redo603\_tmp'  '#ib\_redo604\_tmp'  '#ib\_redo605\_tmp' | '#ib\_redo607\_tmp'  '#ib\_redo608\_tmp'  '#ib\_redo609\_tmp'  '#ib\_redo610\_tmp'  '#ib\_redo611\_tmp'  '#ib\_redo612\_tmp'  '#ib\_redo613\_tmp' |

Each ordinary redo log file is associated with a particular range of LSN values; for example, the following query shows the START\_LSN and END\_LSN values for the active redo log files listed in the previous example:

mysql> **SELECT** **FILE\_NAME,** **START\_LSN,** **END\_LSN** **FROM** **performance\_schema** **.innodb\_redo\_log\_files;**

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

|

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  | | ./#innodb\_redo/#ib\_redo582  ./#innodb\_redo/#ib\_redo583  ./#innodb\_redo/#ib\_redo584  ./#innodb\_redo/#ib\_redo585  ./#innodb\_redo/#ib\_redo586  ./#innodb\_redo/#ib\_redo587  ./#innodb\_redo/#ib\_redo588  ./#innodb\_redo/#ib\_redo589  ./#innodb\_redo/#ib\_redo590  ./#innodb\_redo/#ib\_redo591  ./#innodb\_redo/#ib\_redo592  ./#innodb\_redo/#ib\_redo593  ./#innodb\_redo/#ib\_redo594  ./#innodb\_redo/#ib\_redo595  ./#innodb\_redo/#ib\_redo596  ./#innodb\_redo/#ib\_redo597  ./#innodb\_redo/#ib\_redo598  ./#innodb\_redo/#ib\_redo599  ./#innodb\_redo/#ib\_redo600  ./#innodb\_redo/#ib\_redo601  ./#innodb redo/#ib redo602 |  | 117654982144 |  117658256896  117661531648  117664806400 |  117668081152 |  117671355904 |  117674630656  117677905408  117681180160 |  117684454912 |  117687729664 |  117691004416  117694279168  117697553920 |  117700828672 |  117704103424 |  117707378176  117710652928  117713927680 |  117717202432 |  117720477184 | | 117658256896 |  117661531648  117664806400  117668081152 |  117671355904 |  117674630656 |  117677905408  117681180160  117684454912 |  117687729664 |  117691004416 |  117694279168  117697553920  117700828672 |  117704103424 |  117707378176 |  117710652928  117713927680  117717202432 |  117720477184 |  117723751936 | |

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

| START\_LSN | END\_LSN

| FILE\_NAME

|

|

|

|

|

|

|

|

|

|

|

|

|

When doing a checkpoint, InnoDB stores the checkpoint LSN in the header of the file which contains this LSN. During recovery, all redo log files are checked and recovery starts at the latest checkpoint

LSN.

Several status variables are provided for monitoring the redo log and redo log capacity resize operations; for example, you can query Innodb\_redo\_log\_resize\_status to view the status of a resize operation:

mysql> **SHOW** **STATUS** **LIKE** **'Innodb\_redo\_log\_resize\_status';**

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

| Variable\_name | Value |

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

| Innodb\_redo\_log\_resize\_status | OK |

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

The Innodb\_redo\_log\_capacity\_resized status variable shows the current redo log capacity limit:

mysql> **SHOW** **STATUS** **LIKE** **'Innodb\_redo\_log\_capacity\_resized';**

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

| Variable\_name | Value |

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

| Innodb\_redo\_log\_capacity\_resized | 104857600 |

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

Other applicable status variables include:

• Innodb\_redo\_log\_checkpoint\_lsn

• Innodb\_redo\_log\_current\_lsn

• Innodb\_redo\_log\_flushed\_to\_disk\_lsn

• Innodb\_redo\_log\_logical\_size

• Innodb\_redo\_log\_physical\_size

• Innodb\_redo\_log\_read\_only

• Innodb\_redo\_log\_uuid

Refer to the status variable descriptions for more information.

You can view information about active redo log files by querying the innodb\_redo\_log\_files Performance Schema table. The following query retrieves data from all of the table's columns:

SELECT FILE\_ID, START\_LSN, END\_LSN, SIZE\_IN\_BYTES, IS\_FULL, CONSUMER\_LEVEL

FROM performance\_schema.innodb\_redo\_log\_files;

**Configuring** **Redo** **Log** **Capacity** **(Before** **MySQL** **8.0.30)**

Prior to MySQL 8.0.30, InnoDB creates two redo log files in the data directory by default, named ib\_logfile0 and ib\_logfile1, and writes to these files in a circular fashion.

Modifying redo log capacity requires changing the number or the size of redo log files, or both.

1. Stop the MySQL server and make sure that it shuts down without errors.

2. Edit my.cnf to change the redo log file configuration. To change the redo log file size, configure innodb\_log\_file\_size. To increase the number of redo log files, configure innodb\_log\_files\_in\_group.

3. Start the MySQL server again.

If InnoDB detects that the innodb\_log\_file\_size differs from the redo log file size, it writes a log checkpoint, closes and removes the old log files, creates new log files at the requested size, and opens the new log files.

**Automatic** **Redo** **Log** **Capacity** **Configuration**

When innodb\_dedicated\_server is enabled, InnoDB automatically configures certain InnoDB parameters, including redo log capacity. Automated configuration is intended for MySQL instances that reside on a server dedicated to MySQL, where the MySQL server can use all available system resources. For more information, see Section 15.8.12, “Enabling Automatic Configuration for a Dedicated MySQL Server” .

**Redo** **Log** **Archiving**

Backup utilities that copy redo log records may sometimes fail to keep pace with redo log generation while a backup operation is in progress, resulting in lost redo log records due to those records being overwritten. This issue most often occurs when there is significant MySQL server activity during the backup operation, and the redo log file storage media operates at a faster speed than the backup storage media. The redo log archiving feature, introduced in MySQL 8.0.17, addresses this issue by sequentially writing redo log records to an archive file in addition to the redo log files. Backup utilities can copy redo log records from the archive file as necessary, thereby avoiding the potential loss of data.

If redo log archiving is configured on the server, [MySQL Enterprise Backup](https://dev.mysql.com/doc/mysql-enterprise-backup/8.0/en/), available with the [MySQL](https://www.mysql.com/products/enterprise/) [Enterprise Edition](https://www.mysql.com/products/enterprise/), uses the redo log archiving feature when backing up a MySQL server.

Enabling redo log archiving on the server requires setting a value for the innodb\_redo\_log\_archive\_dirs system variable. The value is specified as a semicolon- separated list of labeled redo log archive directories. The *label:directory* pair is separated by a colon (:). For example:



mysql> SET GLOBAL innodb\_redo\_log\_archive\_dirs='*label1*:*directory\_path1* [;*label2*:*directory\_path2*;…]';

The *label* is an arbitrary identifier for the archive directory. It can be any string of characters, with the exception of colons (:), which are not permitted. An empty label is also permitted, but the colon (:) is still required in this case. A *directory\_path* must be specified. The directory selected for the redo log archive file must exist when redo log archiving is activated, or an error is returned. The path can contain colons (':'), but semicolons (;) are not permitted.

The innodb\_redo\_log\_archive\_dirs variable must be configured before redo log archiving can be activated. The default value is NULL, which does not permit activating redo log archiving.

**Notes**

The archive directories that you specify must satisfy the following requirements. (The requirements are enforced when redo log archiving is activated.):

• Directories must exist. Directories are not created by the redo log archive process. Otherwise, the following error is returned:

ERROR 3844 (HY000): Redo log archive directory

'*directory\_path1* ' does not exist or is not a directory

• Directories must not be world-accessible. This is to prevent the redo log data from being exposed to unauthorized users on the system. Otherwise, the following error is returned:

ERROR 3846 (HY000): Redo log archive directory

'*directory\_path1* ' is accessible to all OS users

• Directories cannot be those defined by datadir, innodb\_data\_home\_dir, innodb\_directories, innodb\_log\_group\_home\_dir, innodb\_temp\_tablespaces\_dir, innodb\_tmpdir innodb\_undo\_directory, or secure\_file\_priv, nor can they be parent directories or subdirectories of those directories. Otherwise, an error similar to the following is returned:

ERROR 3845 (HY000): Redo log archive directory

'*directory\_path1* ' is in, under, or over server directory

'datadir' - '*/path/to/data\_directory*'

When a backup utility that supports redo log archiving initiates a backup, the backup utility activates redo log archiving by invoking the innodb\_redo\_log\_archive\_start() function.

If you are not using a backup utility that supports redo log archiving, redo log archiving can also be activated manually, as shown:

mysql> SELECT innodb\_redo\_log\_archive\_start('*label* ', '*subdir*');

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

| innodb\_redo\_log\_archive\_start('*label* ') |

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

| 0 |

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

Or:

mysql> **DO** **innodb\_redo\_log\_archive\_start('*1abe1*',** **'*subdir*');**

Query OK, 0 rows affected (0.09 sec)

**Note**

The MySQL session that activates redo log archiving (using innodb\_redo\_log\_archive\_start()) must remain open for the duration of the archiving. The same session must deactivate redo log archiving (using

innodb\_redo\_log\_archive\_stop()). If the session is terminated before the redo log archiving is explicitly deactivated, the server deactivates redo log archiving implicitly and removes the redo log archive file.

where *label* is a label defined by innodb\_redo\_log\_archive\_dirs; subdir is an optional argument for specifying a subdirectory of the directory identified by *label* for saving the archive file; it must be a simple directory name (no slash (/), backslash (\), or colon (:) is permitted). subdir can be empty, null, or it can be left out.

Only users with the INNODB\_REDO\_LOG\_ARCHIVE privilege can activate redo log

archiving by invoking innodb\_redo\_log\_archive\_start(), or deactivate it using innodb\_redo\_log\_archive\_stop(). The MySQL user running the backup utility or the MySQL user activating and deactivating redo log archiving manually must have this privilege.

The redo log archive file path is *directory\_identified\_by\_label*/

[*subdir*/]archive .*serverUUID*.000001 .log, where *directory\_identified\_by\_label* is the archive directory identified by the *label* argument for innodb\_redo\_log\_archive\_start().

*subdir* is the optional argument used for innodb\_redo\_log\_archive\_start(). For example, the full path and name for a redo log archive file appears similar to the following:

/*directory\_path*/*subdirectory*/archive.e71a47dc-61f8-11e9-a3cb-080027154b4d.000001.log

After the backup utility finishes copying InnoDB data files, it deactivates redo log archiving by calling the innodb\_redo\_log\_archive\_stop() function.

If you are not using a backup utility that supports redo log archiving, redo log archiving can also be deactivated manually, as shown:

mysql> **SELECT** **innodb\_redo\_log\_archive\_stop();**

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

| innodb\_redo\_log\_archive\_stop() |

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

| 0 |

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

Or:

mysql> **DO** **innodb\_redo\_log\_archive\_stop();**

Query OK, 0 rows affected (0.01 sec)

After the stop function completes successfully, the backup utility looks for the relevant section of redo log data from the archive file and copies it into the backup.

After the backup utility finishes copying the redo log data and no longer needs the redo log archive file, it deletes the archive file.

Removal of the archive file is the responsibility of the backup utility in normal situations. However, if the redo log archiving operation quits unexpectedly before innodb\_redo\_log\_archive\_stop() is called, the MySQL server removes the file.

**Performance** **Considerations**

Activating redo log archiving typically has a minor performance cost due to the additional write activity.

On Unix and Unix-like operating systems, the performance impact is typically minor, assuming there is not a sustained high rate of updates. On Windows, the performance impact is typically a bit higher, assuming the same.

If there is a sustained high rate of updates and the redo log archive file is on the same storage media as the redo log files, the performance impact may be more significant due to compounded write activity.

If there is a sustained high rate of updates and the redo log archive file is on slower storage media than the redo log files, performance is impacted arbitrarily.



Writing to the redo log archive file does not impede normal transactional logging except in the case that the redo log archive file storage media operates at a much slower rate than the redo log file storage media, and there is a large backlog of persisted redo log blocks waiting to be written to the redo log archive file. In this case, the transactional logging rate is reduced to a level that can be managed by the slower storage media where the redo log archive file resides.

**Disabling** **Redo** **Logging**

As of MySQL 8.0.21, you can disable redo logging using the ALTER INSTANCE DISABLE INNODB REDO\_LOG statement. This functionality is intended for loading data into a new MySQL instance. Disabling redo logging speeds up data loading by avoiding redo log writes and doublewrite buffering.

**Warning**

This feature is intended only for loading data into a new MySQL instance. *Do* *not* *disable* *redo* *logging* *on* *a* *production* *system.* It is permitted to shutdown and restart the server while redo logging is disabled, but an unexpected server stoppage while redo logging is disabled can cause data loss and instance corruption.

Attempting to restart the server after an unexpected server stoppage while redo logging is disabled is refused with the following error:

[ERROR] [MY-013598] [InnoDB] Server was killed when Innodb Redo

logging was disabled . Data files could be corrupt . You can try

to restart the database with innodb\_force\_recovery=6

In this case, initialize a new MySQL instance and start the data loading procedure again.

The INNODB\_REDO\_LOG\_ENABLE privilege is required to enable and disable redo logging. The Innodb\_redo\_log\_enabled status variable permits monitoring redo logging status.

Cloning operations and redo log archiving are not permitted while redo logging is disabled and vice versa.

An ALTER INSTANCE [ENABLE |DISABLE] INNODB REDO\_LOG operation requires an exclusive backup metadata lock, which prevents other ALTER INSTANCE operations from executing concurrently. Other ALTER INSTANCE operations must wait for the lock to be released before executing.

The following procedure demonstrates how to disable redo logging when loading data into a new MySQL instance.

1. On the new MySQL instance, grant the INNODB\_REDO\_LOG\_ENABLE privilege to the user account

responsible for disabling redo logging. mysql> GRANT INNODB\_REDO\_LOG\_ENABLE ON \*.\* to 'data\_load\_admin';

2. As the data\_load\_admin user, disable redo logging: mysql> ALTER INSTANCE DISABLE INNODB REDO\_LOG;

3. Check the Innodb\_redo\_log\_enabled status variable to ensure that redo logging is disabled.

mysql> **SHOW** **GLOBAL** **STATUS** **LIKE** **'Innodb\_redo\_log\_enabled';**

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

| Variable\_name | Value |

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

| Innodb\_redo\_log\_enabled | OFF |

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

4. Run the data load operation.

5. As the data\_load\_admin user, enable redo logging after the data load operation finishes:

mysql> ALTER INSTANCE ENABLE INNODB REDO\_LOG;

6. Check the Innodb\_redo\_log\_enabled status variable to ensure that redo logging is enabled.

mysql> **SHOW** **GLOBAL** **STATUS** **LIKE** **'Innodb\_redo\_log\_enabled';**

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

| Variable\_name | Value |

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

| Innodb\_redo\_log\_enabled | ON |

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

**Related** **Topics**

• Redo Log Configuration

• Section 8.5.4, “Optimizing InnoDB Redo Logging”

• Redo Log Encryption

**15.6.6** **Undo** **Logs**

An undo log is a collection of undo log records associated with a single read-write transaction. An undo log record contains information about how to undo the latest change by a transaction to a clustered index record. If another transaction needs to see the original data as part of a consistent read operation, the unmodified data is retrieved from undo log records. Undo logs exist within undo log segments, which are contained within rollback segments. Rollback segments reside in undo tablespaces and in the global temporary tablespace.

Undo logs that reside in the global temporary tablespace are used for transactions that modify data in user-defined temporary tables. These undo logs are not redo-logged, as they are not required for crash recovery. They are used only for rollback while the server is running. This type of undo log benefits performance by avoiding redo logging I/O.

For information about data-at-rest encryption for undo logs, see Undo Log Encryption.

Each undo tablespace and the global temporary tablespace individually support a maximum of 128 rollback segments. The innodb\_rollback\_segments variable defines the number of rollback segments.

The number of transactions that a rollback segment supports depends on the number of undo slots in the rollback segment and the number of undo logs required by each transaction. The number of undo

slots in a rollback segment differs according to InnoDB page size.

|  |  |
| --- | --- |
| **InnoDB** **Page** **Size** | **Number** **of** **Undo** **Slots** **in** **a** **Rollback** **Segment** **(InnoDB** **Page** **Size** **/** **16)** |
| 4096 (4KB) | 256 |
| 8192 (8KB) | 512 |
| 16384 (16KB) | 1024 |
| 32768 (32KB) | 2048 |
| 65536 (64KB) | 4096 |

A transaction is assigned up to four undo logs, one for each of the following operation types:

1. INSERT operations on user-defined tables

2. UPDATE and DELETE operations on user-defined tables

3. INSERT operations on user-defined temporary tables



4. UPDATE and DELETE operations on user-defined temporary tables

Undo logs are assigned as needed. For example, a transaction that performs INSERT, UPDATE, and DELETE operations on regular and temporary tables requires a full assignment of four undo logs. A transaction that performs only INSERT operations on regular tables requires a single undo log.

A transaction that performs operations on regular tables is assigned undo logs from an assigned undo tablespace rollback segment. A transaction that performs operations on temporary tables is assigned undo logs from an assigned global temporary tablespace rollback segment.

An undo log assigned to a transaction remains attached to the transaction for its duration. For example, an undo log assigned to a transaction for an INSERT operation on a regular table is used for all INSERT operations on regular tables performed by that transaction.

Given the factors described above, the following formulas can be used to estimate the number of concurrent read-write transactions that InnoDB is capable of supporting.

**Note**

It is possible to encounter a concurrent transaction limit error before reaching the number of concurrent read-write transactions that InnoDB is capable of supporting. This occurs when a rollback segment assigned to a transaction runs out of undo slots. In such cases, try rerunning the transaction.

When transactions perform operations on temporary tables, the number of concurrent read-write transactions that InnoDB is capable of supporting is constrained by the number of rollback segments allocated to the global temporary tablespace, which is 128 by default.

• If each transaction performs either an INSERT **or** an UPDATE or DELETE operation, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

(innodb\_page\_size / 16) \* innodb\_rollback\_segments \* number of undo tablespaces

• If each transaction performs an INSERT **and** an UPDATE or DELETE operation, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

(innodb\_page\_size / 16 / 2) \* innodb\_rollback\_segments \* number of undo tablespaces

• If each transaction performs an INSERT operation on a temporary table, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

(innodb\_page\_size / 16) \* innodb\_rollback\_segments

• If each transaction performs an INSERT **and** an UPDATE or DELETE operation on a temporary table, the number of concurrent read-write transactions that InnoDB is capable of supporting is:

(innodb\_page\_size / 16 / 2) \* innodb\_rollback\_segments

**15.7** **InnoDB** **Locking** **and** **Transaction** **Model**

To implement a large-scale, busy, or highly reliable database application, to port substantial code from a different database system, or to tune MySQL performance, it is important to understand InnoDB locking and the InnoDB transaction model.

This section discusses several topics related to InnoDB locking and the InnoDB transaction model with which you should be familiar.

• [Section 15.7.1, “InnoDB Locking”](#_bookmark45) describes lock types used by InnoDB.

• [Section 15.7.2, “InnoDB Transaction Model”](#_bookmark46) describes transaction isolation levels and the locking strategies used by each. It also discusses the use of autocommit, consistent non-locking reads, and locking reads.

• Section 15.7.3, “Locks Set by Different SQL Statements in InnoDB” discusses specific types of locks set in InnoDB for various statements.

• Section 15.7.4, “Phantom Rows” describes how InnoDB uses next-key locking to avoid phantom rows.

• Section 15.7.5, “Deadlocks in InnoDB” provides a deadlock example, discusses deadlock detection, and provides tips for minimizing and handling deadlocks in InnoDB.

**15.7.1** **InnoDB** **Locking**

This section describes lock types used by InnoDB.

• [Shared and Exclusive Locks](#_bookmark47)

• [Intention Locks](#_bookmark48)

• [Record Locks](#_bookmark49)

• [Gap Locks](#_bookmark50)

• [Next-Key Locks](#_bookmark51)

• [Insert Intention Locks](#_bookmark52)

• [AUTO-INC Locks](#_bookmark53)

• [Predicate Locks for Spatial Indexes](#_bookmark54)

**Shared** **and** **Exclusive** **Locks**

InnoDB implements standard row-level locking where there are two types of locks, shared (S) locks and exclusive (X) locks.

• A shared (S) lock permits the transaction that holds the lock to read a row.

• An exclusive (X) lock permits the transaction that holds the lock to update or delete a row.

If transaction T1 holds a shared (S) lock on row r, then requests from some distinct transaction T2 for a lock on row r are handled as follows:

• A request by T2 for an S lock can be granted immediately. As a result, both T1 and T2 hold an S lock on r.

• A request by T2 for an X lock cannot be granted immediately.

If a transaction T1 holds an exclusive (X) lock on row r, a request from some distinct transaction T2 for a lock of either type on r cannot be granted immediately. Instead, transaction T2 has to wait for transaction T1 to release its lock on row r.

**Intention** **Locks**

InnoDB supports *multiple* *granularity* *locking* which permits coexistence of row locks and table locks. For example, a statement such as LOCK TABLES ... WRITE takes an exclusive lock (an X lock) on the specified table. To make locking at multiple granularity levels practical, InnoDB uses intention locks. Intention locks are table-level locks that indicate which type of lock (shared or exclusive) a transaction requires later for a row in a table. There are two types of intention locks:

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

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

For example, SELECT ... FOR SHARE sets an IS lock, and SELECT ... FOR UPDATE sets an IX lock.

The intention locking protocol is as follows:

• Before a transaction can acquire a shared lock on a row in a table, it must first acquire an IS lock or stronger on the table.

• Before a transaction can acquire an exclusive lock on a row in a table, it must first acquire an IX lock on the table.

Table-level lock type compatibility is summarized in the following matrix.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | X | 工X | S | 工S |
| X | Conflict | Conflict | Conflict | Conflict |
| IX | Conflict | Compatible | Conflict | Compatible |
| S | Conflict | Conflict | Compatible | Compatible |
| IS | Conflict | Compatible | Compatible | Compatible |

A lock is granted to a requesting transaction if it is compatible with existing locks, but not if it conflicts with existing locks. A transaction waits until the conflicting existing lock is released. If a lock request conflicts with an existing lock and cannot be granted because it would cause deadlock, an error occurs.

Intention locks do not block anything except full table requests (for example, LOCK TABLES ... WRITE). The main purpose of intention locks is to show that someone is locking a row, or going to lock a row in the table.

Transaction data for an intention lock appears similar to the following in SHOW ENGINE INNODB STATUS and InnoDB monitor output:

TABLE LOCK table `test`.`t` trx id 10080 lock mode IX

**Record** **Locks**

A record lock is a lock on an index record. For example, SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE; prevents any other transaction from inserting, updating, or deleting rows where the value of t.c1 is 10.

Record locks always lock index records, even if a table is defined with no indexes. For such cases, InnoDB creates a hidden clustered index and uses this index for record locking. See [Section 15.6.2.1,](#_bookmark1) [“Clustered and Secondary Indexes”](#_bookmark1) .

Transaction data for a record lock appears similar to the following in SHOW ENGINE INNODB STATUS and InnoDB monitor output:

RECORD LOCKS space id 58 page no 3 n bits 72 index `PRIMARY` of table `test`.`t`

trx id 10078 lock\_mode X locks rec but not gap

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 3; compact format; info bits 0

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

1: len 6; hex 00000000274f; asc 'O;;

2: len 7; hex b60000019d0110; asc ;;

**Gap** **Locks**

A gap lock is a lock on a gap between index records, or a lock on the gap before the first or after the last index record. For example, SELECT c1 FROM t WHERE c1 BETWEEN 10 and 20 FOR UPDATE; prevents other transactions from inserting a value of 15 into column t.c1, whether or not there was already any such value in the column, because the gaps between all existing values in the range are locked.

A gap might span a single index value, multiple index values, or even be empty.

Gap locks are part of the tradeoff between performance and concurrency, and are used in some transaction isolation levels and not others.

Gap locking is not needed for statements that lock rows using a unique index to search for a unique row. (This does not include the case that the search condition includes only some columns of a multiple-column unique index; in that case, gap locking does occur.) For example, if the id column has a unique index, the following statement uses only an index-record lock for the row having id value 100 and it does not matter whether other sessions insert rows in the preceding gap:

SELECT \* FROM child WHERE id = 100;

If id is not indexed or has a nonunique index, the statement does lock the preceding gap.

It is also worth noting here that conflicting locks can be held on a gap by different transactions. For example, transaction A can hold a shared gap lock (gap S-lock) on a gap while transaction B holds an exclusive gap lock (gap X-lock) on the same gap. The reason conflicting gap locks are allowed is that if a record is purged from an index, the gap locks held on the record by different transactions must be merged.

Gap locks in InnoDB are “purely inhibitive” , which means that their only purpose is to prevent other transactions from inserting to the gap. Gap locks can co-exist. A gap lock taken by one transaction does not prevent another transaction from taking a gap lock on the same gap. There is no difference between shared and exclusive gap locks. They do not conflict with each other, and they perform the same function.

Gap locking can be disabled explicitly. This occurs if you change the transaction isolation level to [READ](#_bookmark55) [COMMITTED](#_bookmark55). In this case, gap locking is disabled for searches and index scans and is used only for foreign-key constraint checking and duplicate-key checking.

There are also other effects of using the [READ COMMITTED](#_bookmark55) isolation level. Record locks for nonmatching rows are released after MySQL has evaluated the WHERE condition. For UPDATE statements, InnoDB does a “semi-consistent” read, such that it returns the latest committed version to MySQL so that MySQL can determine whether the row matches the WHERE condition of the UPDATE.

**Next-Key** **Locks**

A next-key lock is a combination of a record lock on the index record and a gap lock on the gap before the index record.

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. A next-key lock on an index record also affects the “gap” before that 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.

Suppose that an index contains the values 10, 11, 13, and 20. The possible next-key locks for this index cover the following intervals, where a round bracket denotes exclusion of the interval endpoint and a square bracket denotes inclusion of the endpoint:

(negative infinity, 10]

(10, 11]

(11, 13]

(13, 20]

(20, positive infinity)

For the last interval, the next-key lock locks the gap above the largest value in the index and the “supremum” pseudo-record having a value higher than any value actually in the index. The supremum is not a real index record, so, in effect, this next-key lock locks only the gap following the largest index value.

By default, InnoDB operates in [REPEATABLE READ](#_bookmark56) transaction isolation level. In this case, InnoDB uses next-key locks for searches and index scans, which prevents phantom rows (see Section 15.7.4, “Phantom Rows” ).

Transaction data for a next-key lock appears similar to the following in SHOW ENGINE INNODB STATUS and InnoDB monitor output:

RECORD LOCKS space id 58 page no 3 n bits 72 index `PRIMARY` of table `test`.`t`

trx id 10080 lock\_mode X

Record lock, heap no 1 PHYSICAL RECORD: n\_fields 1; compact format; info bits 0

0: len 8; hex 73757072656d756d; asc supremum;;

Record lock, heap no 2 PHYSICAL RECORD: n\_fields 3; compact format; info bits 0

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

1: len 6; hex 00000000274f; asc 'O;;

2: len 7; hex b60000019d0110; asc ;;

**Insert** **Intention** **Locks**

An insert intention lock is a type of gap lock set by INSERT operations prior to row insertion. 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, respectively, 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.

The following example demonstrates a transaction taking an insert intention lock prior to obtaining an exclusive lock on the inserted record. The example involves two clients, A and B.

Client A creates a table containing two index records (90 and 102) and then starts a transaction that places an exclusive lock on index records with an ID greater than 100. The exclusive lock includes a gap lock before record 102:

mysql> **CREATE** **TABLE** **child** **(id** **int(11)** **NOT** **NULL,** **PRIMARY** **KEY(id))** **ENGINE=InnoDB;**

mysql> **INSERT** **INTO** **child** **(id)** **values** **(90),(102);**

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

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

+-----+

| id |

+-----+

| 102 |

+-----+

Client B begins a transaction to insert a record into the gap. The transaction takes an insert intention lock while it waits to obtain an exclusive lock.

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

mysql> **INSERT** **INTO** **child** **(id)** **VALUES** **(101);**

Transaction data for an insert intention lock appears similar to the following in SHOW ENGINE INNODB STATUS and InnoDB monitor output:

RECORD LOCKS space id 31 page no 3 n bits 72 index `PRIMARY` of table `test`.`child`

trx id 8731 lock\_mode X locks gap before rec **insert** **intention** waiting

Record lock, heap no 3 PHYSICAL RECORD: n\_fields 3; compact format; info bits 0

0: len 4; hex 80000066; asc f;;

1: len 6; hex 000000002215; asc " ;;

2: len 7; hex 9000000172011c; asc r ;;...

**AUTO-INC** **Locks**

An AUTO-INC lock is a special table-level lock taken by transactions inserting into tables with AUTO\_INCREMENT columns. In the simplest case, if one transaction is inserting values into the table, any other transactions must wait to do their own inserts into that table, so that rows inserted by the first transaction receive consecutive primary key values.

The innodb\_autoinc\_lock\_mode variable controls the algorithm used for auto-increment locking. It allows you to choose how to trade off between predictable sequences of auto-increment values and maximum concurrency for insert operations.

For more information, see Section 15.6.1.6, “AUTO\_INCREMENT Handling in InnoDB” .

**Predicate** **Locks** **for** **Spatial** **Indexes**

InnoDB supports SPATIAL indexing of columns containing spatial data (see Section 11.4.9, “Optimizing Spatial Analysis”).

To handle locking for operations involving SPATIAL indexes, next-key locking does not work well to support [REPEATABLE READ](#_bookmark56) or [SERIALIZABLE](#_bookmark57) transaction isolation levels. There is no absolute ordering concept in multidimensional data, so it is not clear which is the “next” key.

To enable support of isolation levels for tables with SPATIAL indexes, InnoDB uses predicate locks. A SPATIAL index contains minimum bounding rectangle (MBR) values, so InnoDB enforces consistent read on the index by setting a predicate lock on the MBR value used for a query. Other transactions cannot insert or modify a row that would match the query condition.

**15.7.2** **InnoDB** **Transaction** **Model**

The InnoDB transaction model aims to combine the best properties of a multi-versioning database with traditional two-phase locking. InnoDB performs locking at the row level and runs queries as nonlocking consistent reads by default, in the style of Oracle. The lock information in InnoDB is stored space- efficiently so that lock escalation is not needed. Typically, several users are permitted to lock every row in InnoDB tables, or any random subset of the rows, without causing InnoDB memory exhaustion.

**15.7.2.1** **Transaction** **Isolation** **Levels**

Transaction isolation is one of the foundations of database processing. Isolation is the I in the acronym ACID; the isolation level is the setting that fine-tunes the balance between performance and reliability, consistency, and reproducibility of results when multiple transactions are making changes and performing queries at the same time.

InnoDB offers all four transaction isolation levels described by the SQL:1992 standard: [READ](#_bookmark58) [UNCOMMITTED](#_bookmark58), [READ COMMITTED](#_bookmark55), [REPEATABLE READ](#_bookmark56), and [SERIALIZABLE](#_bookmark57). The default isolation level for InnoDB is [REPEATABLE READ](#_bookmark56).

A user can change the isolation level for a single session or for all subsequent connections with the SET TRANSACTION statement. To set the server's default isolation level for all connections, use the -- transaction-isolation option on the command line or in an option file. For detailed information about isolation levels and level-setting syntax, see Section 13.3.7, “SET TRANSACTION Statement” .

InnoDB supports each of the transaction isolation levels described here using different locking strategies. You can enforce a high degree of consistency with the default [REPEATABLE READ](#_bookmark56) level, for operations on crucial data where ACID compliance is important. Or you can relax the consistency rules with [READ COMMITTED](#_bookmark55) or even [READ UNCOMMITTED](#_bookmark58), in situations such as bulk reporting where precise consistency and repeatable results are less important than minimizing the amount of overhead for locking. [SERIALIZABLE](#_bookmark57) enforces even stricter rules than [REPEATABLE READ](#_bookmark56), and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

The following list describes how MySQL supports the different transaction levels. The list goes from the most commonly used level to the least used.

• REPEATABLE READ

This is the default isolation level for InnoDB. Consistent reads within the same transaction read the snapshot established by the first read. This means that if you issue several plain (nonlocking) SELECT statements within the same transaction, these SELECT statements are consistent also with respect to each other. See [Section 15.7.2.3, “Consistent Nonlocking Reads”](#_bookmark59) .

For locking reads (SELECT with FOR UPDATE or FOR SHARE), UPDATE, and DELETE statements, locking depends 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, 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”](#_bookmark45) .

• READ COMMITTED

Each consistent read, even within the same transaction, sets and reads its own fresh snapshot. For information about consistent reads, see [Section 15.7.2.3, “Consistent Nonlocking Reads”](#_bookmark59) .

For locking reads (SELECT with FOR UPDATE or FOR SHARE), UPDATE statements, and DELETE statements, InnoDB locks only index records, not the gaps before them, and thus permits the free insertion of new records next to locked records. Gap locking is only used for foreign-key constraint checking and duplicate-key checking.

Because gap locking is disabled, phantom row problems may occur, as other sessions can insert new rows into the gaps. For information about phantom rows, see Section 15.7.4, “Phantom Rows” .

Only row-based binary logging is supported with the READ COMMITTED isolation level. If you use READ COMMITTED with binlog\_format=MIXED, the server automatically uses row-based logging.

Using READ COMMITTED has additional effects:

• For UPDATE or DELETE statements, InnoDB holds locks only for rows that it updates or deletes. Record locks for nonmatching rows are released after MySQL has evaluated the WHERE condition. This greatly reduces the probability of deadlocks, but they can still happen.

• For UPDATE statements, if a row is already locked, InnoDB performs a “semi-consistent” read, returning the latest committed version to MySQL so that MySQL can determine whether the row matches the WHERE condition of the UPDATE. If the row matches (must be updated), MySQL reads the row again and this time InnoDB either locks it or waits for a lock on it.

Consider the following example, beginning with this table:

CREATE TABLE t (a INT NOT NULL, b INT) ENGINE = InnoDB;

INSERT INTO t VALUES (1,2),(2,3),(3,2),(4,3),(5,2);

COMMIT;

In this case, the table has no indexes, so searches and index scans use the hidden clustered index for record locking (see [Section 15.6.2.1, “Clustered and Secondary Indexes”](#_bookmark1)) rather than indexed columns.

Suppose that one session performs an UPDATE using these statements:

# Session A

START TRANSACTION;

UPDATE t SET b = 5 WHERE b = 3;

Suppose also that a second session performs an UPDATE by executing these statements following those of the first session:

# Session B

UPDATE t SET b = 4 WHERE b = 2;

As InnoDB executes each UPDATE, it first acquires an exclusive lock for each row, and then

determines whether to modify it. If InnoDB does not modify the row, it releases the lock. Otherwise,

InnoDB retains the lock until the end of the transaction. This affects transaction processing as follows.

When using the default REPEATABLE READ isolation level, the first UPDATE acquires an x-lock on each row that it reads and does not release any of them:

x-lock(1,2); retain x-lock

x-lock(2,3); update(2,3) to (2,5); retain x-lock

x-lock(3,2); retain x-lock

x-lock(4,3); update(4,3) to (4,5); retain x-lock

x-lock(5,2); retain x-lock

The second UPDATE blocks as soon as it tries to acquire any locks (because first update has retained locks on all rows), and does not proceed until the first UPDATE commits or rolls back:

x-lock(1,2); block and wait for first UPDATE to commit or roll back

If READ COMMITTED is used instead, the first UPDATE acquires an x-lock on each row that it reads and releases those for rows that it does not modify:

x-lock(1,2); unlock(1,2)

x-lock(2,3); update(2,3) to (2,5); retain x-lock

x-lock(3,2); unlock(3,2)

x-lock(4,3); update(4,3) to (4,5); retain x-lock

x-lock(5,2); unlock(5,2)

For the second UPDATE, InnoDB does a “semi-consistent” read, returning the latest committed version of each row that it reads to MySQL so that MySQL can determine whether the row matches the WHERE condition of the UPDATE:

x-lock(1,2); update(1,2) to (1,4); retain x-lock

x-lock(2,3); unlock(2,3)

x-lock(3,2); update(3,2) to (3,4); retain x-lock

x-lock(4,3); unlock(4,3)

x-lock(5,2); update(5,2) to (5,4); retain x-lock

However, if the WHERE condition includes an indexed column, and InnoDB uses the index, only the indexed column is considered when taking and retaining record locks. In the following example, the first UPDATE takes and retains an x-lock on each row where b = 2. The second UPDATE blocks when it tries to acquire x-locks on the same records, as it also uses the index defined on column b.

CREATE TABLE t (a INT NOT NULL, b INT, c INT, INDEX (b)) ENGINE = InnoDB;

INSERT INTO t VALUES (1,2,3),(2,2,4);

COMMIT;

|  |  |
| --- | --- |
| # Session A  START TRANSACTION;  UPDATE t SET b = 3 WHERE  # Session B  UPDATE t SET b = 4 WHERE | b = 2 AND c = 3;  b = 2 AND c = 4; |

The READ COMMITTED isolation level can be set at startup or changed at runtime. At runtime, it can be set globally for all sessions, or individually per session.

• READ UNCOMMITTED

SELECT statements are performed in a nonlocking fashion, but a possible earlier version of a row might be used. Thus, using this isolation level, such reads are not consistent. This is also called a dirty read. Otherwise, this isolation level works like [READ COMMITTED](#_bookmark55).

• SERIALIZABLE

This level is like [REPEATABLE READ](#_bookmark56), but InnoDB implicitly converts all plain SELECT statements to SELECT ... FOR SHARE if autocommit is disabled. If autocommit is enabled, the SELECT is its own transaction. It therefore is known to be read only and can be serialized if performed as a



consistent (nonlocking) read and need not block for other transactions. (To force a plain SELECT to block if other transactions have modified the selected rows, disable autocommit.)

**Note**

As of MySQL 8.0.22, DML operations that read data from MySQL grant tables (through a join list or subquery) but do not modify them do not acquire read locks on the MySQL grant tables, regardless of the isolation level. For more information, see Grant Table Concurrency.

**15.7.2.2** **autocommit,** **Commit,** **and** **Rollback**

In InnoDB, all user activity occurs inside a transaction. If autocommit mode is enabled, each SQL statement forms a single transaction on its own. By default, MySQL starts the session for each new connection with autocommit enabled, so MySQL does a commit after each SQL statement if that statement did not return an error. If a statement returns an error, the commit or rollback behavior depends on the error. See Section 15.21.5, “InnoDB Error Handling” .

A session that has autocommit enabled can perform a multiple-statement transaction by starting it with an explicit START TRANSACTION or BEGIN statement and ending it with a COMMIT or ROLLBACK statement. See Section 13.3.1, “START TRANSACTION, COMMIT, and ROLLBACK Statements” .

If autocommit mode is disabled within a session with SET autocommit = 0, the session always has a transaction open. A COMMIT or ROLLBACK statement ends the current transaction and a new one starts.

If a session that has autocommit disabled ends without explicitly committing the final transaction, MySQL rolls back that transaction.

Some statements implicitly end a transaction, as if you had done a COMMIT before executing the statement. For details, see Section 13.3.3, “Statements That Cause an Implicit Commit” .

A COMMIT means that the changes made in the current transaction are made permanent and become visible to other sessions. A ROLLBACK statement, on the other hand, cancels all modifications made by the current transaction. Both COMMIT and ROLLBACK release all InnoDB locks that were set during the current transaction.

**Grouping** **DML** **Operations** **with** **Transactions**

By default, connection to the MySQL server begins with autocommit mode enabled, which automatically commits every SQL statement as you execute it. This mode of operation might be unfamiliar if you have experience with other database systems, where it is standard practice to issue a sequence of DML statements and commit them or roll them back all together.

To use multiple-statement transactions, switch autocommit off with the SQL statement SET autocommit = 0 and end each transaction with COMMIT or ROLLBACK as appropriate. To leave autocommit on, begin each transaction with START TRANSACTION and end it with COMMIT or ROLLBACK. The following example shows two transactions. The first is committed; the second is rolled back.

$> **mysql** **test**

mysql> **CREATE** **TABLE** **customer** **(a** **INT,** **b** **CHAR** **(20),** **INDEX** **(a));**

Query OK, 0 rows affected (0 .00 sec)

mysql> **--** **Do** **a** **transaction** **with** **autocommit** **turned** **on** **.**

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

Query OK, 0 rows affected (0 .00 sec)

mysql> **INSERT** **INTO** **customer** **VALUES** **(10,** **'Heikki');**

Query OK, 1 row affected (0 .00 sec)

mysql> **COMMIT;**

Query OK, 0 rows affected (0 .00 sec)

mysql> **--** **Do** **another** **transaction** **with** **autocommit** **turned** **off** **.**

transaction, not necessarily to DML statements. If you insert or modify some rows and then commit that transaction, a DELETE or UPDATE statement issued from another concurrent REPEATABLE READ transaction could affect those just-

mysql> **SET** **autocommit=0;**

Query OK, 0 rows affected (0 .00 sec)

mysql> **INSERT** **INTO** **customer** **VALUES** **(15,** **'John');**

Query OK, 1 row affected (0 .00 sec)

mysql> **INSERT** **INTO** **customer** **VALUES** **(20,** **'Paul');**

Query OK, 1 row affected (0 .00 sec)

mysql> **DELETE** **FROM** **customer** **WHERE** **b** **=** **'Heikki';**

Query OK, 1 row affected (0 .00 sec)

mysql> **--** **Now** **we** **undo** **those** **last** **2** **inserts** **and** **the** **delete** **.**

mysql> **ROLLBACK;**

Query OK, 0 rows affected (0 .00 sec)

mysql> **SELECT** **\*** **FROM** **customer;**

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

| a | b |

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

| 10 | Heikki |

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

1 row in set (0 .00 sec)

mysql>

**Transactions** **in** **Client-Side** **Languages**

In APIs such as PHP, Perl DBI, JDBC, ODBC, or the standard C call interface of MySQL, you can send transaction control statements such as COMMIT to the MySQL server as strings just like any other SQL statements such as SELECT or INSERT. Some APIs also offer separate special transaction commit and rollback functions or methods.

**15.7.2.3** **Consistent** **Nonlocking** **Reads**

A consistent read means that InnoDB uses multi-versioning to present to a query a snapshot of the database at a point in time. The query sees the changes made by transactions that committed before that point in time, and no changes made by later or uncommitted transactions. The exception to this rule is that the query sees the changes made by earlier statements within the same transaction. This exception causes the following anomaly: If you update some rows in a table, a SELECT sees the latest version of the updated rows, but it might also see older versions of any rows. If other sessions simultaneously update the same table, the anomaly means that you might see the table in a state that never existed in the database.

If the transaction isolation level is [REPEATABLE READ](#_bookmark56) (the default level), all consistent reads within the same transaction read the snapshot established by the first such read in that transaction. You can get a fresher snapshot for your queries by committing the current transaction and after that issuing new queries.

With [READ COMMITTED](#_bookmark55) isolation level, each consistent read within a transaction sets and reads its own fresh snapshot.

Consistent read is the default mode in which InnoDB processes SELECT statements in [READ](#_bookmark55) [COMMITTED](#_bookmark55) and [REPEATABLE READ](#_bookmark56) isolation levels. A consistent read does not set any locks on the tables it accesses, and therefore other sessions are free to modify those tables at the same time a consistent read is being performed on the table.

Suppose that you are running in the default [REPEATABLE READ](#_bookmark56) isolation level. When you issue a consistent read (that is, an ordinary SELECT statement), InnoDB gives your transaction a timepoint according to which your query sees the database. If another transaction deletes a row and commits after your timepoint was assigned, you do not see the row as having been deleted. Inserts and updates are treated similarly.

 **Note**

 The snapshot of the database state applies to SELECT statements within a

