

Consider increasing innodb\_change\_buffer\_max\_size on a MySQL server with heavy insert, update, and delete activity, where change buffer merging does not keep pace with new change buffer entries, causing the change buffer to reach its maximum size limit.

Consider decreasing innodb\_change\_buffer\_max\_size on a MySQL server with static data used for reporting, or if the change buffer consumes too much of the memory space shared with the buffer pool, causing pages to age out of the buffer pool sooner than desired.

Test different settings with a representative workload to determine an optimal configuration. The innodb\_change\_buffer\_max\_size variable is dynamic, which permits modifying the setting without restarting the server.

**Monitoring** **the** **Change** **Buffer**

The following options are available for change buffer monitoring:

• InnoDB Standard Monitor output includes change buffer status information. To view monitor data, issue the SHOW ENGINE INNODB STATUS statement.

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

Change buffer status information is located under the INSERT BUFFER AND ADAPTIVE HASH INDEX heading and appears similar to the following:

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

INSERT BUFFER AND ADAPTIVE HASH INDEX

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

Ibuf: size 1, free list len 0, seg size 2, 0 merges

merged operations:

insert 0, delete mark 0, delete 0

discarded operations:

insert 0, delete mark 0, delete 0

Hash table size 4425293, used cells 32, node heap has 1 buffer(s)

13577.57 hash searches/s, 202.47 non-hash searches/s

For more information, see Section 15.17.3, “InnoDB Standard Monitor and Lock Monitor Output” .

• The Information Schema INNODB\_METRICS table provides most of the data points found in InnoDB Standard Monitor output plus other data points. To view change buffer metrics and a description of each, issue the following query:

mysql> **SELECT** **NAME,** **COMMENT** **FROM** **INFORMATION\_SCHEMA.INNODB\_METRICS** **WHERE** **NAME** **LIKE** **'%ibuf%'\G**

See Section 15.15.6, “InnoDB INFORMATION\_SCHEMA Metrics Table” .

• The Information Schema INNODB\_BUFFER\_PAGE table provides metadata about each page in the buffer pool, including change buffer index and change buffer bitmap pages. Change buffer pages are identified by PAGE\_TYPE. IBUF\_INDEX is the page type for change buffer index pages, and IBUF\_BITMAP is the page type for change buffer bitmap pages.

**Warning**

Querying the INNODB\_BUFFER\_PAGE table can introduce significant performance overhead. To avoid impacting performance, reproduce the issue you want to investigate on a test instance and run your queries on the test instance.

For example, you can query the INNODB\_BUFFER\_PAGE table to determine the approximate number of IBUF\_INDEX and IBUF\_BITMAP pages as a percentage of total buffer pool pages.

mysql> **SELECT** **(SELECT** **COUNT(\*)** **FROM** **INFORMATION\_SCHEMA.INNODB\_BUFFER\_PAGE**

**WHERE** **PAGE\_TYPE** **LIKE** **'IBUF%')** **AS** **change\_buffer\_pages,**

**(SELECT** **COUNT(\*)** **FROM** **INFORMATION\_SCHEMA** **.INNODB\_BUFFER\_PAGE)** **AS** **total\_pages,**

**(SELECT** **((change\_buffer\_pages/total\_pages)\*100))**

**AS** **change\_buffer\_page\_percentage;**

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

| change\_buffer\_pages | total\_pages | change\_buffer\_page\_percentage |

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

| 25 | 8192 | 0.3052 |

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

For information about other data provided by the INNODB\_BUFFER\_PAGE table, see Section 26.4.2, “The INFORMATION\_SCHEMA INNODB\_BUFFER\_PAGE Table” . For related usage information, see Section 15.15.5, “InnoDB INFORMATION\_SCHEMA Buffer Pool Tables” .

• Performance Schema provides change buffer mutex wait instrumentation for advanced performance monitoring. To view change buffer instrumentation, issue the following query:

mysql> **SELECT** **\*** **FROM** **performance\_schema** **.setup\_instruments**

**WHERE** **NAME** **LIKE** **'%wait/synch/mutex/innodb/ibuf%';**

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

| NAME | ENABLED | TIMED |

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

|

| wait/synch/mutex/innodb/ibuf\_bitmap\_mutex

| wait/synch/mutex/innodb/ibuf\_mutex

|

|

|

YES

YES

YES

YES

|

|

|

YES

|

| wait/synch/mutex/innodb/ibuf\_pessimistic\_insert\_mutex | YES

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

For information about monitoring InnoDB mutex waits, see Section 15.16.2, “Monitoring InnoDB Mutex Waits Using Performance Schema” .

**15.5.3** **Adaptive** **Hash** **Index**

The adaptive hash index enables InnoDB to perform more like an in-memory database on systems with appropriate combinations of workload and sufficient memory for the buffer pool without sacrificing transactional features or reliability. The adaptive hash index is enabled by the innodb\_adaptive\_hash\_index variable, or turned off at server startup by --skip-innodb- adaptive-hash-index.

Based on the observed pattern of searches, a hash index is built using a prefix of the index key. The prefix can be any length, and it may be that only some values in the B-tree appear in the hash index. Hash indexes are built on demand for the pages of the index that are accessed often.

If a table fits almost entirely in main memory, a hash index speeds up queries by enabling direct lookup of any element, turning the index value into a sort of pointer. InnoDB has a mechanism that monitors index searches. If InnoDB notices that queries could benefit from building a hash index, it does so automatically.

With some workloads, the speedup from hash index lookups greatly outweighs the extra work to monitor index lookups and maintain the hash index structure. Access to the adaptive hash index can sometimes become a source of contention under heavy workloads, such as multiple concurrent joins. Queries with LIKE operators and % wildcards also tend not to benefit. For workloads that do not benefit from the adaptive hash index, turning it off reduces unnecessary performance overhead. Because it is difficult to predict in advance whether the adaptive hash index is appropriate for a particular system and workload, consider running benchmarks with it enabled and disabled.

The adaptive hash index feature is partitioned. Each index is bound to a specific partition,

and each partition is protected by a separate latch. Partitioning is controlled by the innodb\_adaptive\_hash\_index\_parts variable. The innodb\_adaptive\_hash\_index\_parts variable is set to 8 by default. The maximum setting is 512.

You can monitor adaptive hash index use and contention in the SEMAPHORES section of SHOW ENGINE INNODB STATUS output. If there are numerous threads waiting on rw-latches created in btr0sea.c, consider increasing the number of adaptive hash index partitions or disabling the adaptive hash index.

For information about the performance characteristics of hash indexes, see Section 8.3.9, “Comparison of B-Tree and Hash Indexes” .

**15.5.4** **Log** **Buffer**

The log buffer is the memory area that holds data to be written to the log files on disk. Log buffer size is defined by the innodb\_log\_buffer\_size variable. The default size is 16MB. The contents of the log buffer are periodically flushed to disk. A large log buffer enables large transactions to run without the need to write redo log data to disk before the transactions commit. Thus, if you have transactions that update, insert, or delete many rows, increasing the size of the log buffer saves disk I/O.

The innodb\_flush\_log\_at\_trx\_commit variable controls how the contents of the log buffer are written and flushed to disk. The innodb\_flush\_log\_at\_timeout variable controls log flushing frequency.

For related information, see Memory Configuration, and Section 8.5.4, “Optimizing InnoDB Redo Logging” .

**15.6** **InnoDB** **On-Disk** **Structures**

This section describes InnoDB on-disk structures and related topics.

**15.6.1** **Tables**

This section covers topics related to InnoDB tables.

**15.6.1.1** **Creating** **InnoDB** **Tables**

InnoDB tables are created using the CREATE TABLE statement; for example:

CREATE TABLE t1 (a INT, b CHAR (20), PRIMARY KEY (a)) ENGINE=InnoDB;

The ENGINE=InnoDB clause is not required when InnoDB is defined as the default storage engine, which it is by default. However, the ENGINE clause is useful if the CREATE TABLE statement is to be replayed on a different MySQL Server instance where the default storage engine is not InnoDB or is unknown. You can determine the default storage engine on a MySQL Server instance by issuing the following statement:

mysql> **SELECT** **@@default\_storage\_engine;**

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

| @@default\_storage\_engine |

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

|  |  |
| --- | --- |
| | InnoDB | | |

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

InnoDB tables are created in file-per-table tablespaces by default. To create an InnoDB table in the InnoDB system tablespace, disable the innodb\_file\_per\_table variable before creating the table. To create an InnoDB table in a general tablespace, use CREATE TABLE ... TABLESPACE syntax. For more information, see Section 15.6.3, “Tablespaces” .

**Row** **Formats**

The row format of an InnoDB table determines how its rows are physically stored on disk. InnoDB supports four row formats, each with different storage characteristics. Supported row formats include REDUNDANT, COMPACT, DYNAMIC, and COMPRESSED. The DYNAMIC row format is the default. For information about row format characteristics, see Section 15.10, “InnoDB Row Formats” .

The innodb\_default\_row\_format variable defines the default row format. The row format of a table can also be defined explicitly using the ROW\_FORMAT table option in a CREATE TABLE or ALTER TABLE statement. See Defining the Row Format of a Table.

**Primary** **Keys**

It is recommended that you define a primary key for each table that you create. When selecting primary key columns, choose columns with the following characteristics:

• Columns that are referenced by the most important queries.

• Columns that are never left blank.

• Columns that never have duplicate values.

• Columns that rarely if ever change value once inserted.

For example, in a table containing information about people, you would not create a primary key on

(firstname, lastname) because more than one person can have the same name, a name column may be left blank, and sometimes people change their names. With so many constraints, often there is not an obvious set of columns to use as a primary key, so you create a new column with a numeric ID to serve as all or part of the primary key. You can declare an auto-increment column so that ascending values are filled in automatically as rows are inserted:

# The value of ID can act like a pointer between related items in different tables.

CREATE TABLE t5 (id INT AUTO\_INCREMENT, b CHAR (20), PRIMARY KEY (id));

# The primary key can consist of more than one column. Any autoinc column must come first.

CREATE TABLE t6 (id INT AUTO\_INCREMENT, a INT, b CHAR (20), PRIMARY KEY (id,a));

For more information about auto-increment columns, see Section 15.6.1.6, “AUTO\_INCREMENT Handling in InnoDB” .

Although a table works correctly without defining a primary key, the primary key is involved with many aspects of performance and is a crucial design aspect for any large or frequently used table. It is recommended that you always specify a primary key in the CREATE TABLE statement. If you create the table, load data, and then run ALTER TABLE to add a primary key later, that operation is much slower than defining the primary key when creating the table. For more information about primary keys, see Section 15.6.2.1, “Clustered and Secondary Indexes” .

**Viewing** **InnoDB** **Table** **Properties**

To view the properties of an InnoDB table, issue a SHOW TABLE STATUS statement:

mysql> **SHOW** **TABLE** **STATUS** **FROM** **test** **LIKE** **'t%'** **\G;**

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

Name: t1

Engine: InnoDB

Version: 10

Row\_format: Dynamic

Rows: 0

Avg\_row\_length: 0

Data\_length: 16384

Max\_data\_length: 0

Index\_length: 0

Data\_free: 0

Auto\_increment: NULL

Create\_time: 2021-02-18 12:18:28

Update\_time: NULL

Check\_time: NULL

Collation: utf8mb4\_0900\_ai\_ci

Checksum: NULL

Create\_options:

Comment:

For information about SHOW TABLE STATUS output, see Section 13.7.7.38, “SHOW TABLE STATUS

Statement” .

You can also access InnoDB table properties by querying the InnoDB Information Schema system tables:

mysql> **SELECT** **\*** **FROM** **INFORMATION\_SCHEMA.INNODB\_TABLES** **WHERE** **NAME='test/t1'** **\G**

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

TABLE\_ID: 1144

NAME: test/t1

FLAG: 33

N\_COLS: 5

SPACE: 30

ROW\_FORMAT: Dynamic

ZIP\_PAGE\_SIZE: 0

SPACE\_TYPE: Single

INSTANT\_COLS: 0

For more information, see Section 15.15.3, “InnoDB INFORMATION\_SCHEMA Schema Object

Tables” .

**15.6.1.2** **Creating** **Tables** **Externally**

There are different reasons for creating InnoDB tables externally; that is, creating tables outside of the data directory. Those reasons might include space management, I/O optimization, or placing tables on

a storage device with particular performance or capacity characteristics, for example. InnoDB supports the following methods for creating tables externally:

• [Using the DATA DIRECTORY Clause](#_bookmark1)

• Using CREATE TABLE ... TABLESPACE Syntax

• Creating a Table in an External General Tablespace

**Using** **the** **DATA** **DIRECTORY** **Clause**

You can create an InnoDB table in an external directory by specifying a DATA DIRECTORY clause in the CREATE TABLE statement.

CREATE TABLE t1 (c1 INT PRIMARY KEY) DATA DIRECTORY = '*/external/directory*';

The DATA DIRECTORY clause is supported for tables created in file-per-table tablespaces. Tables are implicitly created in file-per-table tablespaces when the innodb\_file\_per\_table variable is enabled, which it is by default.

mysql> **SELECT** **@@innodb\_file\_per\_table;**

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

| @@innodb\_file\_per\_table |

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

| 1 |

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

For more information about file-per-table tablespaces, see Section 15.6.3.2, “File-Per-Table Tablespaces” .

When you specify a DATA DIRECTORY clause in a CREATE TABLE statement, the table's data file (*table\_name*.ibd) is created in a schema directory under the specified directory.

As of MySQL 8.0.21, tables and table partitions created outside of the data directory using the DATA DIRECTORY clause are restricted to directories known to InnoDB. This requirement permits database administrators to control where tablespace data files are created and ensures that data files can be found during recovery (see Tablespace Discovery During Crash Recovery). Known directories are those defined by the datadir, innodb\_data\_home\_dir, and innodb\_directories variables. You can use the following statement to check those settings:

mysql> SELECT @@datadir,@@innodb\_data\_home\_dir,@@innodb\_directories;

If the directory you want to use is unknown, add it to the innodb\_directories setting before you create the table. The innodb\_directories variable is read-only. Configuring it requires restarting