 Section 24.6.2, “Partitioning Limitations Relating to Storage Engines” , for more

information.

• [Adjusting Memory Usage for MyISAM and InnoDB](#_bookmark1)

• [Handling Too-Long Or Too-Short Transactions](#_bookmark2)

• [Handling Deadlocks](#_bookmark3)

• [Storage Layout](#_bookmark4)

• [Converting an Existing Table](#_bookmark5)

• [Cloning the Structure of a Table](#_bookmark6)

• [Transferring Data](#_bookmark7)

• [Storage Requirements](#_bookmark8)

• [Defining Primary Keys](#_bookmark9)

• [Application Performance Considerations](#_bookmark10)

• [Understanding Files Associated with InnoDB Tables](#_bookmark11)

**Adjusting** **Memory** **Usage** **for** **MyISAM** **and** **InnoDB**

As you transition away from MyISAM tables, lower the value of the key\_buffer\_size configuration option to free memory no longer needed for caching results. Increase the value of the innodb\_buffer\_pool\_size configuration option, which performs a similar role of allocating cache memory for InnoDB tables. The InnoDB buffer pool caches both table data and index data, speeding up lookups for queries and keeping query results in memory for reuse. For guidance regarding buffer pool size configuration, see Section 8.12.3.1, “How MySQL Uses Memory” .

**Handling** **Too-Long** **Or** **Too-Short** **Transactions**

Because MyISAM tables do not support transactions, you might not have paid much attention to the autocommit configuration option and the COMMIT and ROLLBACK statements. These keywords are important to allow multiple sessions to read and write InnoDB tables concurrently, providing substantial scalability benefits in write-heavy workloads.

While a transaction is open, the system keeps a snapshot of the data as seen at the beginning of the transaction, which can cause substantial overhead if the system inserts, updates, and deletes millions of rows while a stray transaction keeps running. Thus, take care to avoid transactions that run for too long:

• If you are using a mysql session for interactive experiments, always COMMIT (to finalize the

changes) or ROLLBACK (to undo the changes) when finished. Close down interactive sessions rather than leave them open for long periods, to avoid keeping transactions open for long periods by accident.

• Make sure that any error handlers in your application also ROLLBACK incomplete changes or COMMIT completed changes.

• ROLLBACK is a relatively expensive operation, because INSERT, UPDATE, and DELETE operations are written to InnoDB tables prior to the COMMIT, with the expectation that most changes are committed successfully and rollbacks are rare. When experimenting with large volumes of data, avoid making changes to large numbers of rows and then rolling back those changes.

• When loading large volumes of data with a sequence of INSERT statements, periodically COMMIT the results to avoid having transactions that last for hours. In typical load operations for data warehousing, if something goes wrong, you truncate the table (using TRUNCATE TABLE) and start

over from the beginning rather than doing a ROLLBACK.

The preceding tips save memory and disk space that can be wasted during too-long transactions. When transactions are shorter than they should be, the problem is excessive I/O. With each COMMIT, MySQL makes sure each change is safely recorded to disk, which involves some I/O.

• For most operations on InnoDB tables, you should use the setting autocommit=0. From an efficiency perspective, this avoids unnecessary I/O when you issue large numbers of consecutive INSERT, UPDATE, or DELETE statements. From a safety perspective, this allows you to issue a ROLLBACK statement to recover lost or garbled data if you make a mistake on the mysql command line, or in an exception handler in your application.

• autocommit=1 is suitable for InnoDB tables when running a sequence of queries for generating reports or analyzing statistics. In this situation, there is no I/O penalty related to COMMIT or ROLLBACK, and InnoDB can automatically optimize the read-only workload.

• If you make a series of related changes, finalize all the changes at once with a single COMMIT at the end. For example, if you insert related pieces of information into several tables, do a single COMMIT after making all the changes. Or if you run many consecutive INSERT statements, do a single COMMIT after all the data is loaded; if you are doing millions of INSERT statements, perhaps split up the huge transaction by issuing a COMMIT every ten thousand or hundred thousand records, so the transaction does not grow too large.

• Remember that even a SELECT statement opens a transaction, so after running some report or debugging queries in an interactive mysql session, either issue a COMMIT or close the mysql session.

For related information, see Section 15.7.2.2, “autocommit, Commit, and Rollback” .

**Handling** **Deadlocks**

You might see warning messages referring to “deadlocks” in the MySQL error log, or the output of SHOW ENGINE INNODB STATUS. A deadlock is not a serious issue for InnoDB tables, and often does not require any corrective action. When two transactions start modifying multiple tables, accessing the tables in a different order, they can reach a state where each transaction is waiting for the other and neither can proceed. When deadlock detection is enabled (the default), MySQL immediately detects this condition and cancels (rolls back) the “smaller” transaction, allowing the other to proceed. If deadlock detection is disabled using the innodb\_deadlock\_detect configuration option, InnoDB relies on the innodb\_lock\_wait\_timeout setting to roll back transactions in case of a deadlock.

Either way, your applications need error-handling logic to restart a transaction that is forcibly cancelled due to a deadlock. When you re-issue the same SQL statements as before, the original timing issue no longer applies. Either the other transaction has already finished and yours can proceed, or the other transaction is still in progress and your transaction waits until it finishes.

If deadlock warnings occur constantly, you might review the application code to reorder the

SQL operations in a consistent way, or to shorten the transactions. You can test with the innodb\_print\_all\_deadlocks option enabled to see all deadlock warnings in the MySQL error

log, rather than only the last warning in the SHOW ENGINE INNODB STATUS output. For more information, see Section 15.7.5, “Deadlocks in InnoDB” .

**Storage** **Layout**

To get the best performance from InnoDB tables, you can adjust a number of parameters related to storage layout.

When you convert MyISAM tables that are large, frequently accessed, and hold vital data, investigate and consider the innodb\_file\_per\_table and innodb\_page\_size variables, and the ROW\_FORMAT and KEY\_BLOCK\_SIZE clauses of the CREATE TABLE statement.

During your initial experiments, the most important setting is innodb\_file\_per\_table. When this setting is enabled, which is the default, new InnoDB tables are implicitly created in file-per-table

tablespaces. In contrast with the InnoDB system tablespace, file-per-table tablespaces allow disk space to be reclaimed by the operating system when a table is truncated or dropped. File-per-table tablespaces also support DYNAMIC and COMPRESSED row formats and associated features such as table compression, efficient off-page storage for long variable-length columns, and large index prefixes. For more information, see Section 15.6.3.2, “File-Per-Table Tablespaces” .

You can also store InnoDB tables in a shared general tablespace, which support multiple tables and all row formats. For more information, see Section 15.6.3.3, “General Tablespaces” .

**Converting** **an** **Existing** **Table**

To convert a non- InnoDB table to use InnoDB use ALTER TABLE:

ALTER TABLE *table\_name* ENGINE=InnoDB;

**Cloning** **the** **Structure** **of** **a** **Table**

You might make an InnoDB table that is a clone of a MyISAM table, rather than using ALTER TABLE to perform conversion, to test the old and new table side-by-side before switching.

Create an empty InnoDB table with identical column and index definitions. Use SHOW CREATE TABLE *table\_name*\G to see the full CREATE TABLE statement to use. Change the ENGINE clause to

ENGINE=INNODB.

**Transferring** **Data**

To transfer a large volume of data into an empty InnoDB table created as shown in the previous section, insert the rows with INSERT INTO *innodb\_table* SELECT \* FROM *myisam\_table* ORDER BY *primary\_key\_columns*.

You can also create the indexes for the InnoDB table after inserting the data. Historically, creating new secondary indexes was a slow operation for InnoDB, but now you can create the indexes after the data is loaded with relatively little overhead from the index creation step.

If you have UNIQUE constraints on secondary keys, you can speed up a table import by turning off the uniqueness checks temporarily during the import operation:

SET unique\_checks=0;

*...* *import* *operation* *...*

SET unique\_checks=1;

For big tables, this saves disk I/O because InnoDB can use its change buffer to write secondary index records as a batch. Be certain that the data contains no duplicate keys. unique\_checks permits but does not require storage engines to ignore duplicate keys.

For better control over the insertion process, you can insert big tables in pieces:

INSERT INTO newtable SELECT \* FROM oldtable

WHERE yourkey > *something* AND yourkey <= *somethingelse*;

After all records are inserted, you can rename the tables.

During the conversion of big tables, increase the size of the InnoDB buffer pool to reduce disk I/O. Typically, the recommended buffer pool size is 50 to 75 percent of system memory. You can also increase the size of InnoDB log files.

**Storage** **Requirements**

If you intend to make several temporary copies of your data in InnoDB tables during the conversion process, it is recommended that you create the tables in file-per-table tablespaces so that you can reclaim the disk space when you drop the tables. When the innodb\_file\_per\_table configuration option is enabled (the default), newly created InnoDB tables are implicitly created in file-per-table tablespaces.

Whether you convert the MyISAM table directly or create a cloned InnoDB table, make sure that you have sufficient disk space to hold both the old and new tables during the process. **InnoDB** **tables** **require** **more** **disk** **space** **than** **MyISAM** **tables.** If an ALTER TABLE operation runs out of space, it starts a rollback, and that can take hours if it is disk-bound. For inserts, InnoDB uses the insert buffer to merge secondary index records to indexes in batches. That saves a lot of disk I/O. For rollback, no such mechanism is used, and the rollback can take 30 times longer than the insertion.

In the case of a runaway rollback, if you do not have valuable data in your database, it may be advisable to kill the database process rather than wait for millions of disk I/O operations to complete. For the complete procedure, see Section 15.21.3, “Forcing InnoDB Recovery” .

**Defining** **Primary** **Keys**

The PRIMARY KEY clause is a critical factor affecting the performance of MySQL queries and the space usage for tables and indexes. The primary key uniquely identifies a row in a table. Every row in the table should have a primary key value, and no two rows can have the same primary key value.

These are guidelines for the primary key, followed by more detailed explanations.

• Declare a PRIMARY KEY for each table. Typically, it is the most important column that you refer to in WHERE clauses when looking up a single row.

• Declare the PRIMARY KEY clause in the original CREATE TABLE statement, rather than adding it later through an ALTER TABLE statement.

• Choose the column and its data type carefully. Prefer numeric columns over character or string ones.

• Consider using an auto-increment column if there is not another stable, unique, non-null, numeric column to use.

• An auto-increment column is also a good choice if there is any doubt whether the value of the primary key column could ever change. Changing the value of a primary key column is an expensive

operation, possibly involving rearranging data within the table and within each secondary index.

Consider adding a primary key to any table that does not already have one. Use the smallest practical numeric type based on the maximum projected size of the table. This can make each row slightly more compact, which can yield substantial space savings for large tables. The space savings are multiplied if the table has any secondary indexes, because the primary key value is repeated in each secondary index entry. In addition to reducing data size on disk, a small primary key also lets more data fit into the buffer pool, speeding up all kinds of operations and improving concurrency.

If the table already has a primary key on some longer column, such as a VARCHAR, consider adding a new unsigned AUTO\_INCREMENT column and switching the primary key to that, even if that column is not referenced in queries. This design change can produce substantial space savings in the secondary indexes. You can designate the former primary key columns as UNIQUE NOT NULL to enforce the same constraints as the PRIMARY KEY clause, that is, to prevent duplicate or null values across all those columns.

If you spread related information across multiple tables, typically each table uses the same column for its primary key. For example, a personnel database might have several tables, each with a primary key of employee number. A sales database might have some tables with a primary key of customer number, and other tables with a primary key of order number. Because lookups using the primary key are very fast, you can construct efficient join queries for such tables.

If you leave the PRIMARY KEY clause out entirely, MySQL creates an invisible one for you. It is a 6- byte value that might be longer than you need, thus wasting space. Because it is hidden, you cannot refer to it in queries.

**Application** **Performance** **Considerations**

The reliability and scalability features of InnoDB require more disk storage than equivalent MyISAM tables. You might change the column and index definitions slightly, for better space utilization, reduced

I/O and memory consumption when processing result sets, and better query optimization plans making efficient use of index lookups.

If you set up a numeric ID column for the primary key, use that value to cross-reference with related values in any other tables, particularly for join queries. For example, rather than accepting a country name as input and doing queries searching for the same name, do one lookup to determine the country ID, then do other queries (or a single join query) to look up relevant information across several tables. Rather than storing a customer or catalog item number as a string of digits, potentially using up several bytes, convert it to a numeric ID for storing and querying. A 4-byte unsigned INT column can index over 4 billion items (with the US meaning of billion: 1000 million). For the ranges of the different integer types, see Section 11.1.2, “Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT” .

**Understanding** **Files** **Associated** **with** **InnoDB** **Tables**

InnoDB files require more care and planning than MyISAM files do.

• You must not delete the ibdata files that represent the InnoDB system tablespace.

• Methods of moving or copying InnoDB tables to a different server are described in Section 15.6.1.4, “Moving or Copying InnoDB Tables” .

**15.6.1.6** **AUTO\_INCREMENT** **Handling** **in** **InnoDB**

InnoDB provides a configurable locking mechanism that can significantly improve scalability and performance of SQL statements that add rows to tables with AUTO\_INCREMENT columns. To use the AUTO\_INCREMENT mechanism with an InnoDB table, an AUTO\_INCREMENT column must be defined as the first or only column of some index such that it is possible to perform the equivalent of an indexed SELECT MAX(*ai\_col*) lookup on the table to obtain the maximum column value. The index is not required to be a PRIMARY KEY or UNIQUE, but to avoid duplicate values in the AUTO\_INCREMENT column, those index types are recommended.

This section describes the AUTO\_INCREMENT lock modes, usage implications of different AUTO\_INCREMENT lock mode settings, and how InnoDB initializes the AUTO\_INCREMENT counter.

• [InnoDB AUTO\_INCREMENT Lock Modes](#_bookmark12)

• InnoDB AUTO\_INCREMENT Lock Mode Usage Implications

• InnoDB AUTO\_INCREMENT Counter Initialization

• Notes

**InnoDB** **AUTO\_INCREMENT** **Lock** **Modes**

This section describes the AUTO\_INCREMENT lock modes used to generate auto-increment values, and how each lock mode affects replication. The auto-increment lock mode is configured at startup using the innodb\_autoinc\_lock\_mode variable.

The following terms are used in describing innodb\_autoinc\_lock\_mode settings:

• “INSERT-like” statements

All statements that generate new rows in a table, including INSERT, INSERT ... SELECT, REPLACE, REPLACE ... SELECT, and LOAD DATA. Includes “simple-inserts” , “bulk-inserts” , and “mixed-mode” inserts.

• “Simple inserts”

Statements for which the number of rows to be inserted can be determined in advance (when the statement is initially processed). This includes single-row and multiple-row INSERT and REPLACE statements that do not have a nested subquery, but not INSERT ... ON DUPLICATE KEY

UPDATE.