A *strong* feature of MySQL is the ability to specify a Type when creating a new table. MySQL supports a number of table types, distinguished by a variety of properties. The three most important types are MyISAM, InnoDB, and HEAP.

This section provides a brief look at the different table types recognized by MySQL as well as some of their properties and when the use of one type or another is appropriate.

### Setting/Changing Engine

When you create a new table, you can specify which storage engine to use by adding an `ENGINE` or `TYPE` table option to the `CREATE TABLE` statement.

If no `ENGINE` is specified, then MySQL server uses either MyISAM or InnoDB based on its configuration, normally MyISAM.

**Code Sample:**

```sql
CREATE TABLE table1 (item1 INT) ENGINE = INNODB;
CREATE TABLE table2 (item2 INT) TYPE = MEMORY;
```

*Note:* The older term `TYPE` is deprecated, though it is supported as a synonym for `ENGINE` for backward compatibility.

**Tip:** To change the default, use `--default-storage-engine` or `--default-table-type` server startup option or in the MySQL configuration file, or use:

**Code Sample:**

```sql
SET storage_engine=MYISAM;
CREATE TABLE t1(i1 INT);
SET table_type=InnoDB;
CREATE TABLE t2(i2 INT);
```
To convert a table from one storage engine to another, use an ALTER TABLE as shown:

**Code Sample:**

```
ALTER TABLE t2 ENGINE = MYISAM;
```

*Warning:* If you try to use a storage engine that is not compiled in or that is compiled in but deactivated, MySQL instead creates a table using the default storage engine. This behavior is convenient when you want to copy tables between MySQL servers that support different storage engines. A warning is generated whenever a storage engine is automatically changed.

*Tip:* In a replication setup, the master server may use transactional storage engines, but the slave servers may use only non-transactional storage engines for greater speed.

A database may contain tables of different types. That is, tables need not all be created with the same storage engine.

**MyISAM Tables**

The MyISAM table type should be preferred as it is mature, stable, and simple to manage. MyISAM supports three different storage formats. Two of them, fixed and dynamic format, are chosen automatically depending on the column types chosen. The third, compressed format, can be created with the `myisampack` utility.

Static format is the default for MyISAM tables. It is used when the table contains no variable-length columns (`VARCHAR`, `VARBINARY`, `BLOB` or `TEXT`). Each row is stored using a fixed number of bytes, usually requiring more disk space than dynamic-format tables.

If the table contains a `VARCHAR`, `TEXT` or `BLOB` field or if the table was created with the `ROW_FORMAT=DYNAMIC` table option, MySQL automatically selects Dynamic table type.
You can check or repair MyISAM tables with the mysqlcheck client or myisamchk utility. You can also compress MyISAM tables with myisampack to reduce storage space.

You can use OPTIMIZE TABLE or myisamchk -r to defragment a table.

**Full-Text Search on MyISAM Tables**

Searching for text is useful and can take a long time running on large tables if using LIKE SQL operator to search for a word (or more) in character strings:

```sql
SELECT * FROM film_review WHERE review_text LIKE '%story%'
OR review_text LIKE '%stories%';
```

Searching for several words in several columns can get even worse - as all the records of the table must be read using several character string comparisons A traditional index will not help much here. And Users have become accustomed to using an Internet search engine to input a number of search terms without having to deal with complex search criteria.

To process such queries efficiently, we will use a *full-text index*. This is a particular type of index, one that creates a special searchable or indexed list of all words that appear in a column of a table.

**Creating a Full-Text Index**

To supplement an existing table with a full-text index, you may specify any number of TEXT and (VAR)CHAR columns as shown:

```sql
ALTER TABLE tablename ADD FULLTEXT(column1, column2, ...);
```

*Code Sample:*

```sql
ALTER TABLE film_review ADD FULLTEXT(review_title, review_text);
```

Fulltext indexes can also be defined when tables are created:
CREATE TABLE film_info (  
film_id INT NOT NULL PRIMARY KEY,  
info_text VARCHAR(10000),  
FULLTEXT (info_text));  

Using Full-Text Search

For a full-text search, the SQL expression `MATCH AGAINST` is used:

```
SELECT columns, MATCH(column-list) AGAINST('word-list') FROM tablename  
WHERE MATCH(column-list) AGAINST('word-list') > 0.001
```

The result set here is all data records where search fields contain at least one of the words in the given list. The list of columns in `MATCH` must correspond exactly to the one with which the index was generated.

`MATCH` returns as result a floating-point number whose magnitude reflects the relevance of the result. If no words are found or if the search criteria appear in very many records and are therefore ignored, then `MATCH` returns 0.

Code Sample:
```
SELECT * FROM film_review  
WHERE MATCH(review_title,review_text) AGAINST('song soundtrack') > 0.001;
```

The expression `MATCH > 0.005` excludes results where match is insignificant indicated by `MATCH()` returning very small values.

A `MATCH` expression can be used to order results. The following query returns the five best results. The condition must be formulated with `HAVING`:

Code Sample:
```
SELECT *, MATCH(review_title,review_text) AGAINST('story soundtrack') AS match_strength  
FROM film_review  
HAVING match_strength > 0.001  
ORDER BY match_strength DESC  
LIMIT 10;
```

Fulltext Features and Limitations
- Full-text search works better with larger tables than with smaller.
- Full-text search is based on entire words, alterations such as plurals are considered a different word. So to find relevant records, variants have to be specified - AGAINST('store stores')
  - A word in full-text search is considered to be a character string composed of letters, numbers, and the characters ' and_, and may exclude words like C++.
  - Words must be at least four characters in length dropping words like BOY, though there is a work around via the setting
    \texttt{ft_min_word_len = 3}
  .

- Full-text search is case-insensitive, there is no case distinction in the search criteria in AGAINST.

- The order of words in search criteria is irrelevant.
- Full-text search is currently available only for MyISAM tables.
- The creation or rebuilding of a full-text index on large tables is relatively slow, and troublesome for frequently updated text columns.

\section*{InnoDB Tables}

MySQL supports InnoDB engine as an alternative format to MyISAM

The InnoDB table driver has been an integral component of MySQL since version 3.23.34. The development of the table driver and its commercial support come from the independent company Innobase (see http://www.innodb.com). InnoDB offers the following additional functions:

- \textit{Transactions}: Database operations in InnoDB tables can be executed as transactions. This allows you to execute several logically connected SQL commands as a single entity. If an error occurs during execution, then all of the commands (not only the one during which the error occurred) are nullified. In addition, transactions offer other advantages that improve the security of database applications. Transactions can be executed in all four isolation levels of the ANSI-SQL/92 standard (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE).
- \textit{Row Level Locking}: In implementing transactions, the InnoDB table driver uses internal row level locking. This means that during a transaction, the entire table does not have to be blocked for access by other users (which is the case for a MyISAM table during a LOCK TABLE command), but only the data records are actually affected. If many users are simultaneously making changes on a large table, row level locking can bring about an
enormous advantage in efficiency.

- **Deadlocks**: The InnoDB table driver automatically recognizes deadlocks (that is, the condition in which two processes mutually block each other) and in such a case, terminates one of the two processes automatically.

- **Foreign Key Constraints**: For defined relations between tables, the InnoDB table driver automatically ensures that the referential integrity of the table is preserved.

- **Crash Recovery**: After a crash, InnoDB tables are automatically and very quickly returned to a consistent state assuming the OS file system is intact.

*Note*: The InnoDB storage engine is enabled by default. To block InnoDB tables, add the `skip-innodb` to options.

### InnoDB - Limitations and Drawbacks

- **Tablespace Administration**: Each MyISAM table is stored in its own file. The InnoDB table driver stores all data and indexes in a tablespace, comprising one or more files, that forms a sort of virtual file system and requiring some administration overhead. Also, it is not feasible to copy an InnoDB table by simply copying its file.

- **Storage Requirement**: The storage requirements for InnoDB tables are much greater than those for equivalent MyISAM tables (up to twice as big).

- **Full-Text Index**: For InnoDB tables one cannot use a full-text index.

- **COUNT Problem**: On account of open transactions, it is relatively difficult for the InnoDB table driver to determine the number of records in a table. Therefore, a `SELECT COUNT(*)` FROM InnoDB TABLE is much slower than with MyISAM tables.

- **Locking**: InnoDB uses its own locking algorithms in executing transactions, which is an advantage as they block only individual records and not the entire table, but there are overheads and complexities.


*Warning*: The mysql tables for managing MySQL access privileges cannot be transformed into InnoDB tables. They must remain in MyISAM format.
InnoDB Disk Management

Two important disk-based resources managed by the InnoDB storage engine are its tablespace data files and its log files.

*Note:* If you specify no InnoDB configuration options, MySQL creates an auto-extending 10MB data file named `ibdata1` and two 5MB log files named `ib_logfile0` and `ib_logfile1` in the MySQL data directory. To get good performance, you should explicitly provide InnoDB parameters as discussed in the following examples. Naturally, you should edit the settings to suit your hardware and requirements.

The examples shown here are representative.

To set up the InnoDB tablespace files, use the `innodb_data_file_path` option in the `[mysqld]` section of the options file. The value of `innodb_data_file_path` should be a list of one or more data file specifications. If you name more than one data file, separate them by semicolon.

```
innodb_data_home_dir=/dir_spec
innodb_data_file_path=datafile_spec1[:datafile_spec2]...
... where ...
datafile_spec ::= file_name:file_size [:autoextend[:max:max_file_size]]
```

*Note:* The autoextend and other attributes apply only to the last data file spec.

The `innodb_data_home_dir` option is to explicitly specify a location for InnoDB tablespace files instead of default data directory.
The setting below configures a single auto-extending 10MB data file named ibdata1. InnoDB creates it in the MySQL data directory as no location is given.

```
[mysqld]
innoDB_data_file_path=ibdata1:10M:autoextend
```

A tablespace containing a fixed-size 100MB data file named tsdata1 and a 100MB auto-extending file named tsdata2 that can grow up to 500MB:

```
[mysqld]
innoDB_data_file_path=tsdata1:100M;tsdata2:100M:autoextend:max:500M
```

If you specify the autoextend option for the last data file, InnoDB extends the data file if it runs out of free space in the tablespace. The increment is 8MB at a time by default. It can be modified by changing the `innodb_autoextend_increment` system variable.

We can use absolute paths for the data files as well.

```
[mysqld]
innoDB_data_home_dir=
innoDB_data_file_path=/ibdata/ibdata1:50M;/ibdata/ibdata2:50M:autoextend
```

Preventing Problems

- InnoDB does not create directories, so make sure that the /ibdata and log file directories exist before starting the server.

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

- The data files must be less than 2GB in some filesystems. The combined size of the log files must be less than 4GB. The combined size of data files must be at least 10MB.

- When you create an InnoDB tablespace for the first time, it is best that you start the MySQL server from the command prompt. InnoDB then prints the information about the database creation to the screen, so you can see what is happening. If you do not send server output to the screen, check the server's error log to see what InnoDB prints during the startup process.
- For sanity, place InnoDB options in the [mysqld] group of any option file that your server reads when it starts.

**Moving Files**

You cannot freely move .ibd files between database directories as you can with MyISAM table files. This is because the table definition that is stored in the InnoDB shared tablespace includes the database name, and because InnoDB must preserve the consistency of transaction IDs and log sequence numbers.

To move an .ibd file and the associated table from one database to another, use a RENAME TABLE statement:

**Code Sample:**

RENAME TABLE sakila.film_detail TO sakilakubili.film_detail;

**FOREIGN KEY Constraints in InnoDB**

One very important and critically useful feature InnoDB supports are Foreign Key constraints, that allow for tables to be linked at storage level.

The syntax for a foreign key constraint definition in InnoDB looks like this:

```
Syntax [CONSTRAINT constraint_name] FOREIGN KEY [index_name] (index_col_name, ...)
REFERENCES tbl_name (index_col_name, ...)
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

`index_name` represents a foreign key ID. If given, this is ignored if an index for the foreign key is defined explicitly. Otherwise, if InnoDB creates an index for the foreign key, it uses `index_name` for the index name.

Foreign keys definitions are subject to the following conditions:
- Both tables must be InnoDB tables and they must not be TEMPORARY tables.
- Corresponding columns in the foreign key and the referenced key must have similar internal data types inside InnoDB so that they can be compared without a type conversion. The size and sign of integer types must be the same. The length of string types need not be the same. For non-binary (character) string columns, the character set and collation must be the same.
- In the referencing table, there must be an index where the foreign key columns are listed as the first columns in the same order. Such an index is created on the referencing table automatically if it does not exist. index_name, if given, is used as described previously.
- In the referenced table, there must be an index where the referenced columns are listed as the first columns in the same order.
- Index prefixes on foreign key columns are not supported. One consequence of this is that BLOB and TEXT columns cannot be included in a foreign key, because indexes on those columns must always include a prefix length.
- If the CONSTRAINT symbol clause is given, the symbol value must be unique in the database. If the clause is not given, InnoDB creates the name automatically.

InnoDB rejects any INSERT or UPDATE operation that attempts to create a foreign key value in a child table if there is no a matching candidate key value in the parent table.

For UPDATE or DELETE statements on key values in the parent table with matching rows in the child table, InnoDB behavior is dependent on the referential action using UPDATE and ON DELETE subclauses of the FOREIGN KEY clause, as shown:

- CASCADE:

Delete or update the row from the parent table and automatically delete or update the matching rows in the child table.
- **SET NULL:**

Delete or update the row from the parent table and set the foreign key column or columns in the child table to NULL.

Ensure that you have not declared the columns in the child table as NOT NULL.

- **NO ACTION:**

An attempt to delete or update a primary key value is not allowed to proceed if there is a related foreign key value in the referenced table.

- **RESTRICT:**

Rejects the delete or update operation for the parent table, so NO ACTION and RESTRICT are the same.

- **SET DEFAULT:**

MySQL parser allows this action, but InnoDB rejects table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET DEFAULT clauses, probably for a future version.

InnoDB automatically creates indexes on foreign keys so that foreign key checks can be fast and not require a table scan.

InnoDB does not check foreign key constraints on those foreign key or referenced key values that contain a NULL column.

Currently, triggers are not activated by cascaded foreign key actions.
InnoDB allows you to add a new foreign key constraint to a table by adding constraints or columns via ALTER TABLE. Remember to create the required indexes first.

You can also add a self-referential foreign key constraint to a table using ALTER TABLE.

InnoDB also supports the use of ALTER TABLE to drop foreign keys:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

If the FOREIGN KEY clause included a CONSTRAINT name when you created the foreign key, you can refer to that name to drop the foreign key. Otherwise, the `fk_symbol` value is internally generated by InnoDB when the foreign key is created.

**Warning:** If ALTER TABLE for an InnoDB table results in changes to column values (for example, loss of data), InnoDB’s FOREIGN KEY constraint checks do not notice possible violations.

InnoDB returns a table's foreign key definitions as part of the output of the SHOW CREATE TABLE statement:

```
SHOW CREATE TABLE tbl_name;
```

mysqldump also produces correct definitions of tables to the dump file, and does not forget about the foreign keys.

You can also display the foreign key constraints and fk_symbols for a table:

```
SHOW TABLE STATUS FROM db_name LIKE 'tbl_name';
SHOW TABLE STATUS FROM sakila LIKE 'rental';
```

**Code Sample:**
The foreign key constraints are listed in the Comment column of the output.

**Checking Foreign Keys**

When performing foreign key checks, InnoDB sets shared row-level locks on child or parent records it has to look at. InnoDB checks foreign key constraints immediately; the check is not deferred to transaction commit.

To make it easier to reload dump files for tables that have foreign key relationships, mysqldump automatically includes a statement in the dump output to set FOREIGN_KEY_CHECKS to 0. This avoids problems with tables having to be reloaded in a particular order when the dump is reloaded. It is also possible to set this variable manually:

**Code Sample:**

```sql
SET FOREIGN_KEY_CHECKS = 0;
SOURCE data_dump_file_name;
SET FOREIGN_KEY_CHECKS = 1;
```

This allows you to import the tables in any order if the dump file contains tables that are not correctly ordered for foreign keys and also speeds up the import operation by ignoring foreign key constraints.

However, even if FOREIGN_KEY_CHECKS=0, InnoDB does not allow the creation of a foreign key constraint where a column references a non-matching column type. Also, if an InnoDB table has foreign key constraints, `ALTER TABLE` cannot be used to change the table to use another storage engine. To alter the storage engine, you must drop any foreign key constraints first.

InnoDB does not allow you to drop a table that is referenced by a FOREIGN KEY constraint, unless you do:

```sql
SET FOREIGN_KEY_CHECKS=0
```

When you drop a table, the constraints that were defined in its create statement are also dropped.
Choosing - MyISAM or InnoDB?

A database can contain both MyISAM and InnoDB tables, allowing optimal table driver for each table, depending on its content and usage. Application and database performance depends considerably on hardware, MySQL settings and design. It is best to try out a few options before finalizing any engine type for critical-path tables.

Transactional tables offer some benefits over non-transactional tables:

- Safety: On a database problem, it is possible to get data back, either by automatic recovery or from a backup plus the transaction log.
- Plan units of work: You can combine statements logically and accept or reject them all together in a transaction.
- Better Failure Handling: On failures in non-transactional tables, all changes made so far are permanently written to database, many times leading to inconsistencies.
- Better Concurrency: Transactional tables provide higher and more efficient concurrency for high-updates concurrently with high-reads.
- Avoid Table Locking: InnoDB avoids LOCK TABLE commands that normally hurt concurrent updates.

Note: Within a mixed table-type transaction, changes to non-transaction-safe tables are committed immediately and cannot be rolled back.

Non-transaction-safe tables have several advantages of their own, all of which occur because there is no transaction overhead:

- Faster access:

MyISAM tables are to be recommended whenever you want to manage tables in the most space and time-efficient way possible.

- Lower disk space requirements
- Use lesser memory and other resources required to perform updates

HEAP or MEMORY Tables
HEAP tables exist only in RAM (not on the hard drive). They use a hash index, which results in particularly fast access to individual data records. HEAP tables are often used as temporary tables. See the next section for more about this topic.

In comparison to normal tables, HEAP tables present a large number of functional restrictions, of which we mention here only the most important: No xxxTEXT or xxxBLOB data types can be used. Records can be searched only with = or ?‡" (and not with <, >, <=, or >=). AUTO_INCREMENT is not supported. Indexes can be set up only for NOT NULL columns.

HEAP tables should be used whenever relatively small data sets are to be managed with maximal speed. Since HEAP tables are stored exclusively in RAM, they disappear as soon as MySQL is terminated. The maximum size of a HEAP table is determined in the MySQL configuration file by the parameter `max_heap_table_size`.

**Temporary Tables**

With all of the table types listed above there exists the possibility of creating a table on a temporary basis. Such tables are automatically deleted as soon as the link with MySQL is terminated. Furthermore, temporary tables are invisible to other MySQL links (so that it is possible for two users to employ temporary tables with the same name without running into trouble).

Temporary tables are not a separate table type unto themselves, but rather a variant of the types that we have been describing. Temporary tables are often created automatically by MySQL in order to assist in the execution of SELECT queries.

Temporary tables are not stored in the same directory as the other MySQL tables, but in a special temporary directory (under Windows it is usually called C:\WindowsTemp, while under Unix it is generally /tmp or /var/tmp or /usr/tmp). The directory can be set at MySQL startup.

**MERGE Tables**

The MERGE storage engine, also known as the MRG_MyISAM engine, is a collection of
identical MyISAM tables that can be used as one. Identical means that all tables have identical column and index information.

**Advantage of using MERGE tables:**

1. Manage similar but usually disconnected tables, such as logs for different periods.
2. Faster performance: Split large tables into smaller tables, even across databases.
3. Easier management: On-the-fly combining/disassociating and repair of component tables

4.

**Disadvantages of MERGE tables:**

1. Use only identical MyISAM tables for a MERGE table.
2. A number of MyISAM features in MERGE tables cannot be used, such as FULLTEXT indexes.
3. Key reads are slower. The MERGE storage engine issues a read on all underlying tables to check which one most closely match a given key.


**Code Sample:**

```sql
CREATE TABLE table1 (id1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
key1 CHAR(2) DEFAULT 'T1',
desc1 CHAR(20)) ENGINE=MyISAM;

CREATE TABLE table2 (id2 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
key2 CHAR(2) DEFAULT 'T2',
desc2 CHAR(20)) ENGINE=MyISAM;

INSERT INTO table1 (desc1) VALUES ('Data11'),('Data12'),('Data13');
INSERT INTO table2 (desc2) VALUES ('Data21'),('Data22'),('Data23');

CREATE TABLE merged_table (merged_id INT NOT NULL AUTO_INCREMENT,
merged_who CHAR(2) DEFAULT 'MT',
merged_desc CHAR(20),
) ENGINE=MyISAM;
```
INDEX(merged_id)
) ENGINE=MERGE UNION=(table1,table2) INSERT_METHOD=LAST;

INSERT INTO merged_table (merged_desc) VALUES ('MergedData1'), ('MergedData2'), ('MergedData3');

SELECT * FROM merged_table;
SELECT * FROM table1;
SELECT * FROM table2;

DROP TABLE merged_table, table1, table2;

Code Explanation

1. Create two MyISAM tables.
2. Create a merged table on these two tables.
3. Add data to the two base tables and the merged table.
4. View data from all three tables to see how data is managed in the underlying tables.

Other Table Types

MySQL recognizes a variety of other table types, of which those listed here are the most important variants. Note that these table types are available only in the Max version or self-compiled version of MySQL. You can determine which table types your MySQL version supports with the command SHOW ENGINES.

ARCHIVE or Compressed Tables (since MySQL 4.1)

This table type is designed for the archiving and logging of large data sets. The advantage of this table type is that the records are immediately compressed when they are stored so large amounts of data without indexes can be stored with a very small footprint.

Some key characteristics ARCHIVE Tables of are:

- ARCHIVE tables make sense only if the records are not to be altered. (INSERT is permitted, but UPDATE and DELETE are not allowed.).
- Rows are compressed as they are inserted.
- ARCHIVE tables cannot be indexed. For each SELECT command, therefore, all the records will be read (complete table scan). So use this table type only if you expect to access the data relatively rarely.
- On retrieval, rows are uncompressed and there is no row cache.
The number of rows in ARCHIVE tables reported by SHOW TABLE STATUS is always accurate.

*Note:* Several SELECT statements during insertion can deteriorate compression, so bulk or delayed inserts may be preferred. Also, to achieve better compression, use OPTIMIZE TABLE or REPAIR TABLE.

### CSV Type Tables (since MySQL 4.1)

Records from CSV tables are saved as text files with comma-separated values. For example, "123"," I am a character string".

For a CSV table, the storage engine creates:

- A table format file - `<table-name>` and a `.frm` extension.
- A plain text data file - `<table-name>` and a `.frm` extension. When you store data into the table, the storage engine saves it into the data file in comma-separated values format.

#### Code Sample:

```sql
CREATE TABLE csv_data ( 
i INT NOT NULL, c CHAR(10) NOT NULL
) ENGINE = CSV;

INSERT INTO csv_data VALUES(1,'record one'),(2,'record two');

SELECT * FROM csv_data;
```

**Code Explanation**

The contents of `csv_data.CSV` file in the database directory (for the table above) will read:

"1","record one"
"2","record two"

This CSV format can be read, and even written, by spreadsheet applications such as Microsoft
Excel or StarOffice Calc.

The CSV storage engine does not support indexing.

**NDB or Cluster Tables (since MySQL 4.1)**

The NDB table type belongs with the MySQL cluster functions, which are integrated into the MySQL Max version. (NDB stands for network database.) This table type is transactions-capable and is most suitable for databases that are distributed among a large number of computers. However, the use of this table type requires that first a number of MySQL Max installations be specially configured for cluster operation. Detailed information can be found at [http://dev.mysql.com/doc/mysql/en/ndbcluster.html](http://dev.mysql.com/doc/mysql/en/ndbcluster.html).

**FEDERATED or External Tables (since MySQL 5.0)**

This table type enables access to tables in an external database. The database system can be located, for example, on another computer in the local network. At present, the external database must be a MySQL database using the MySQL C Client API. Though perhaps in the future, MySQL will allow connection with other database systems.

When you create a FEDERATED table, the server creates a table format file in the database directory. The file begins with the table name and has an .frm extension. No other files are created, because the actual data is in a remote table.

There are some restrictions in accessing FEDERATED tables: Neither transactions nor query optimization with Query Cache are possible. The structure of external tables cannot be changed (though the records can be). In other words, ALTER TABLE is not permitted, while INSERT, UPDATE, and DELETE are.

*Tip:* Further information on all the MySQL table types can be found in the MySQL documentation at [http://dev.mysql.com/doc/mysql/en/storage-engines.html](http://dev.mysql.com/doc/mysql/en/storage-engines.html).

**BDB Tables**
BDB or BerkeleyDB tables were historically the first transactions-capable MySQL table type. Now that the InnoDB table driver has matured, there is not much reason to use BDB tables and BDB support is expected to phase out with MySQL v5.1.

Table Files Storage

For new tables, MySQL always creates an .frm file to hold the table and column definitions. The table's index and data may be stored in one or more other files, depending on the storage engine. The server creates the .frm file above the storage engine level. Individual storage engines create any additional files required for the tables that they manage.

You can specify the location for database files at MySQL startup. (Under Unix/Linux /var/lib/mysql is frequently used, while under Windows it is usually \C:Programs\MySQL\MySQL Server n\ndata.) All further specifications are relative to this directory.

A description of each table is saved in a *.frm file. The *.frm files are located in directories whose names correspond to the name of the database: data/dbname/tablename.frm. This file contains the table schema (data types of the columns, etc.).

An additional file, db.opt, is stored in the database directory, which relates to the entire database: data/dbname/db.opt. This file contains the database settings.

For each MyISAM table, two additional files are created: data/dbname/tablename.MYD, with MyISAM table data, and data/dbname/tablename.MYI, with MyISAM indexes (all indexes of the table).

Each MyISAM table is stored on disk in three files. The files have names that begin with the table name and have an extension to indicate the file type. An .frm file stores the table format. The data file has an .MYD (MYData) extension. The index file has an .MYI (MYIndex) extension.
InnoDB tables are stored in individual files for each table or else collectively, in the so-called tablespace (depending on whether innodb_file_per_table is specified in the MySQL configuration file). The location and name of the tablespace are also governed by configuration settings. In current MySQL installations, the default is data/dbname/tablename.ibd for InnoDB table data (data and indexes), and data/ibdata1, -2, -3 for the tablespace and undo logs, and data/ib_logfile0, -1, -2 for InnoDB logging data.

If triggers are defined for the tables, then their code is currently stored in a file data/dbname/tablename.TRG.

For a MERGE table, MySQL creates two files on disk. The files have names that begin with the table name and have an extension to indicate the file type. An .frm file stores the table format, and an .MRG file contains the names of the tables that should be used as one. The tables do not have to be in the same database as the MERGE table itself.

It is not unexpected that some of these storage rules may change in future MySQL versions.

Syntax  MySQL Table Types Conclusion

In this lesson of the MySQL tutorial, we reviewed Table types available in MySQL.

To continue to learn MySQL go to the top of this page and click on the next lesson in this MySQL Tutorial's Table of Contents.