

# **MySQL 5.6 Database Administrator**

---

## Course Contents

### Day 1

Total Hours : 08

Mysql Overview and MySQL Architecture(Theory : 45 Min , Hands on :25 Min )

- MySQL Overview, Products, Services
- MySQL Services and Support
- Supported Operating Services
- MySQL Certification Program
- Training Curriculum Paths
- MySQL Documentation Resources
- Describe the client/server model
- Understand communication protocols
- Understand how the server supports storage engines
- Explain the basics of how MySQL uses memory and disk space
- Choose between types of MySQL distributions
- Install the MySQL Server
- Describe the MySQL Server installation file structure
- Start and stop the MySQL server
- Run multiple MySQL servers on a single host

Server Configuration and Client Tools( Theory : 40 min, Handon : 35 Min)

- Set up MySQL server configuration files
- Explain the purpose of dynamic server variables
- Review the server status variables available
- Configure operational characteristics of the MySQL server
- Describe the available log files
- Explain binary logging
- Describe the available clients for administrative tasks
- Use MySQL administrative clients

- Use the mysql command line clients
- Use the mysqladmin for administrative tasks
- Use the MySQLWorkbench
- Describe available MySQL tools
- List the available APIs (drivers and connectors)

Obtaining Metadata( Theory 30 : Min, Handon : 20)

- List the various metadata access methods available
- Recognize the structure of the INFORMATION\_SCHEMA database schema
- Use the available commands to view metadata
- Describe differences between SHOW statements and INFORMATION\_SCHEMA tables
- Use the mysqlshow client program
- Use INFORMATION\_SCHEMA to create shell commands and SQL statements

Transaction, Locking and Innodb Storage Engine( Theory : 45 Min, Handon : 35 Min)

- Use transaction control statement to run multiple SQL statements concurrently
- Explain the ACID properties
- Describe the transaction isolation levels
- Use locking to protect transactions
- Describe the InnoDB storage engine
- Set the storage engine to InnoDB
- Illustrate the InnoDB tablespace storage system
- Efficiently configure the tablespace
- Use foreign keys to attain referential integrity
- Explain InnoDB locking

User Management(Theory : 45 Min, Hand On : 35)

- Depict the user connection and query process
- List requirements for user authentication
- Use SHOW PROCESSLIST to show which threads are running
- Create, modify and drop user accounts

- List requirements for user authorization
- Describe the levels of access privileges for users
- List the types of privileges
- Grant, modify and revoke user privileges

Security( Theory 30 Min, Hands On : 25 )

- Recognize common security risks
- Describe security risks specific to the MySQL installation
- List security problems and counter-measures for network, operating system, filesystem and users
- Protect your data
- Use SSL for secure MySQL server connections
- Explain how SSH enables a secure remote connection to the MySQL server
- Find additional information for common security issues

Table Maintenance and Exporting-Importing(Theory :25, Handson : 25)

- Recognize types of table maintenance operations
- Execute SQL statements for table maintenance
- Client and utility programs for table maintenance
- Maintain tables according to specific storage engines
- Exporting Data using SQL
- Importing Data using SQL

Day 2:

Programming Inside MySQL( Theory : 40 Hands on : 20)

- Creating and executing Stored Routines
- Describing stored routine execution security
- Creating and executing triggers
- Creating, altering and dropping events
- Explaining event execution scheduling

MySQL Backup and Recovery( Theory 40 Hands On :35)

- Describing backup basics
- Types of backups
- Backup tools and utilities
- Making binary and text backups
- Role of log and status files in backups
- Data Recovery

Introduction to Performance Tuning( Theory 45 Hands On 40)

- Factors that affects Performance
- Using EXPLAIN to Analyze Queries
- Using Procedure Analyze for Schema Design
- General Table Optimizations
- Monitoring status variables that affect performance
- Setting and Interpreting MySQL server Variables
- Performance Schema
- Top Status Variable
- Top System Variables
- Important Tools for Performance Tuning

Introduction to Replication( Theory 60 : Hand ON :60)

- Describing MySQL Replication
- Managing the MySQL Binary Log
- Explaining MySQL replication threads and files
- Setting up a MySQL Replication Environment
- Monitoring MySQL Replication
- Troubleshooting MySQL Replication
- Replication with Global Transaction Identifiers (GTIDs)

Introduction to Mysql Cluster( Theory 60, Handson : 60)

- What is a Cluster
- Key benefits of a Cluster
- Architecture of Cluster

- Installation and Configuration of Cluster
- Using Tool to monitor Cluster
- Troubleshooting of the Cluster

Conclusion

## **Chapter 1**

Mysql Overview and MySQL Architecture

MySQL is a known RDBMS which is :

- The world's most popular open-source database
- Over 15 million estimated active installations
- The M of the LAMP stack
- Used by 9 of the top 10 websites in the world
- Embedded by over 3,000 ISVs and OEMs
- The leading database in the cloud
- Highly popular on social media (Facebook, Twitter, and so on)

MySQL is the world's most popular open-source database and the best database for web-based applications, powering leading websites worldwide including Facebook, Twitter, and YouTube. MySQL is also widely used by enterprise and governmental organizations for web applications as well as custom enterprise applications and data marts. Those use cases are complementary to the deployments of Oracle Database, which are typically found in enterprise resource planning (ERP) applications and other high-end enterprise solutions.

As an extremely popular choice for an embedded database, MySQL is distributed by thousands of independent software vendors (ISVs) and original equipment manufacturers (OEMs). These include software vendors such as Sage (which embeds MySQL in accounting applications) and eClinicalWorks (which ships MySQL as part of medical practice management software). MySQL is also integrated in security solutions delivered by companies such as Cisco and Checkpoint and in numerous telecom offerings (including Alcatel Lucent and Avaya).

Finally, MySQL has become the leading database in the cloud, offered by the majority of cloud service providers and powering many software-as-a-service (SaaS) applications.

MySQL Classic Edition is well-suited for embedded and read-intensive, non-OLTP applications.

MySQL Standard and Enterprise Editions are well suited for read-intensive and OLTP applications that require high performance, high availability, and consistent crash recovery.

In addition to the Community edition, OEMs can license or subscribe to all editions, and end users can subscribe to Standard, Enterprise, and Cluster Carrier Grade editions.

The commercial editions build on each other:

- Standard Edition includes Classic Edition plus the additional features listed in the slide.
- Enterprise Edition includes Standard Edition plus the additional features listed in the slide.
- Carrier Grade Edition includes Enterprise Edition plus the additional features listed in the slide.

MySQL Enterprise Edition includes the most comprehensive set of advanced features, management tools, and technical support to achieve the highest levels of MySQL scalability, security, reliability, and up time. It reduces the risk, cost, and complexity in developing, deploying, and managing business-critical MySQL applications.

MySQL Workbench is available in both GPL and commercial editions. Details about these tools are presented later in this course.

MySQL connectors provide connectivity to the MySQL server for client programs. APIs provide low-level access to the MySQL protocol and MySQL resources. Both the connectors and the APIs enable you to connect and execute MySQL statements from another language or environment.

Third-party connectors that MySQL supports:

- PHP: mysqli, ext/mysqli, PDO\_MYSQLND, PHP\_MYSQLND
- Perl: DBD::mysql
- Python: MySQLdb
- Ruby: DBD::MySQL, ruby-mysql
- C++ Wrapper: For MySQL C API (MySQL++)

The embedded MySQL server library (libmysqld) is also supported. libmysqld makes it possible to run a full-featured MySQL server inside a client application. The main benefits are increased speed and simpler management for embedded applications.

You can download connectors and their documentation from the following page:

<http://mysql.com/products/connector>

For more information about connectors, see the MySQL Reference Manual:

<http://dev.mysql.com/doc/mysql/en/connectors-apis.html>

## **Community Support**

- Mailing lists
  - Forums
  - Community articles
  - PlanetMySQL blogs
  - Twitter
  - Physical and virtual events, including:
    - Developer days
    - MySQL Tech Tours
    - Webinars
  - Bug tracking
  - Launchpad repositories
- 
- Mailing lists (<http://lists.mysql.com>)
  - Forums (<http://forums.mysql.com>)
  - Community articles (<http://dev.mysql.com/tech-resources/articles>)
  - Planet MySQL blogs (<http://planet.mysql.com>)
  - Twitter ([http://twitter.com/mysql\\_community](http://twitter.com/mysql_community) and <http://twitter.com/MySQL>)
  - Physical and virtual events (<http://www.mysql.com/news-and-events>)
  - Bug tracking (<http://bugs.mysql.com>)

- Launchpad repositories

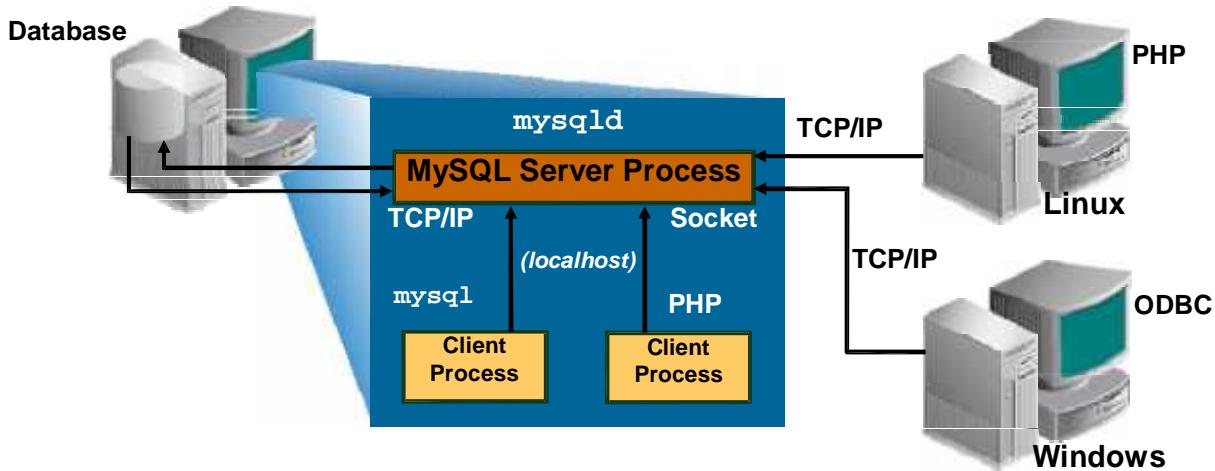
## **MySQL-Supported Operating Systems**

### **MySQL:**

- Provides control and flexibility for users
- Supports multiple commodity platforms, including:
  - Windows (x86, x86\_64)
  - Linux (x86, x86\_64, IA64)
  - Oracle Solaris (SPARC, x86\_64, x86)
  - Mac OS X (x86, x86\_64)

# MySQL Architecture

## MySQL client/server model



A MySQL installation has the following required architectural components: the MySQL server, client programs, and MySQL non-client programs. A central program acts as a server, and the client programs connect to the server to make data requests.

MySQL client/server communication is not limited to environments where all computers run the same operating system.

- Client programs can connect to the server running on the same host or on a different host.
- Client/server communication can occur in environments where computers run different operating systems.

**Note:** Any information in this course that does not apply to all operating systems is identified as platform-specific. Information that works for Linux generally applies for all UNIX-like operating systems.

# Client Programs

- Connect to the MySQL server to retrieve, modify, add, or remove data.
- Use these client programs to perform the following actions:
  - **mysql**: Issue queries and view results.
  - **mysqladmin**: Administer the server.
  - **mysqlcheck**: Check the integrity of database tables.
  - **mysqldump**: Create logical backups.
  - **mysqlimport**: Import text data files.
  - **mysqlshow**: Show database, table, and column information.
  - **mysqlslap**: Emulate client load.
- Use MySQL Workbench for database management.

These programs are executed from the prompt of a command interpreter:

```
shell> mysql [options]
```

The **mysql** client program is commonly known as the *command-line interface* (CLI).

For more information about client programs, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/programs-client.html>.

MySQL Workbench is a GUI tool that enables you to:

- Model a database
- Perform database queries
- Perform administration tasks

# Administrative and Utility Programs

- Access data files directly without using a client to connect to the server.
- Use non-client programs.
  - **innoschecksum**: Check an InnoDB tablespace file offline.
  - **mysqldumpslow**: Summarize the slow query log files.
  - **mysqlbinlog**: Display the binary log files.
- Some of the programs have requirements that must be met prior to running:
  - Shut down the server.
  - Back up your current tables.
- Review program requirements prior to implementation.

These programs are also executed from the prompt of a command interpreter.

`mysqldumpslow` is a Perl script.

There are many more programs available. To avoid loss or corruption of data, some of the programs require that you shut down the server and/or make a back up of your current tables prior to execution.

# Server Process

- Is the database server program called **mysqld**
- Is not the same as a “host”
- Is a single process and is multithreaded
- Manages access to databases on disk and in memory
- Supports simultaneous client connections
- Supports multiple storage engines
- Supports both transactional and nontransactional tables
- Uses memory in the form of:
  - Caching
  - Buffering

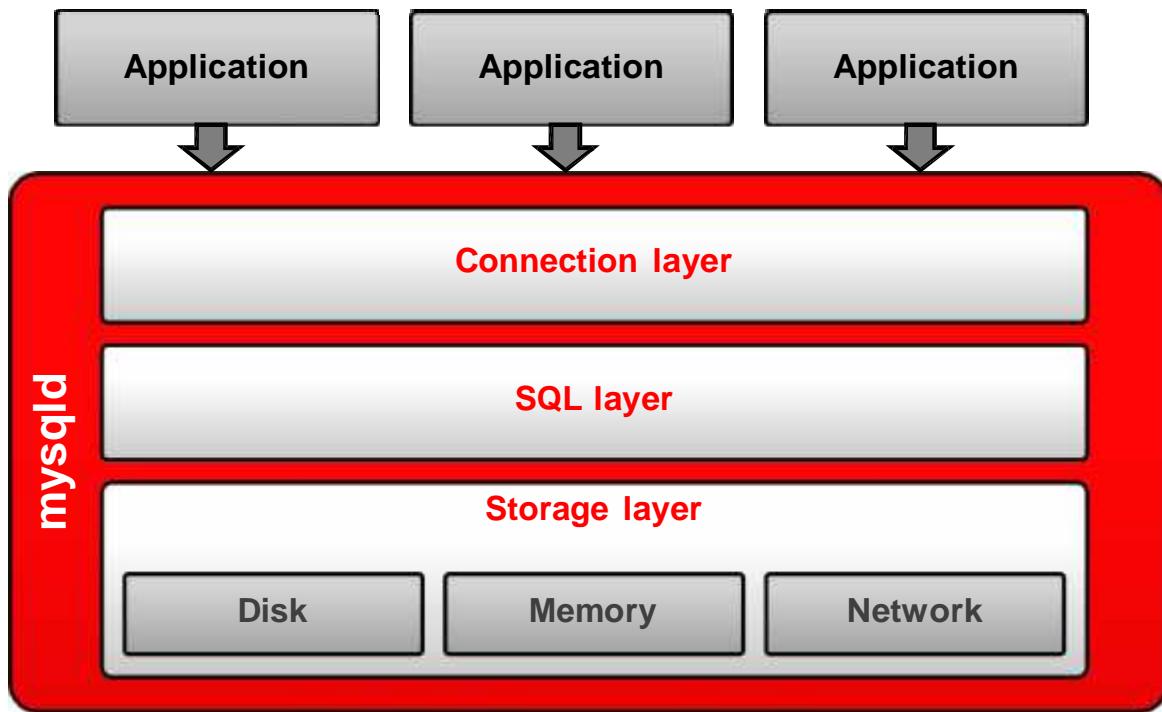
Note the difference between a server and a host:

- **Server:** A software program (`mysqld`) with a version number and a list of features
- **Host:** The physical machine on which the server program runs, which includes the following:
  - Its hardware configuration
  - The operating system running on the machine
  - Its network addresses

Multiple `mysqld` instances can run simultaneously on one host.

The configuration of the MySQL server evolves from one product version to the next. Always consult the product documentation for the most up-to-date configuration information.

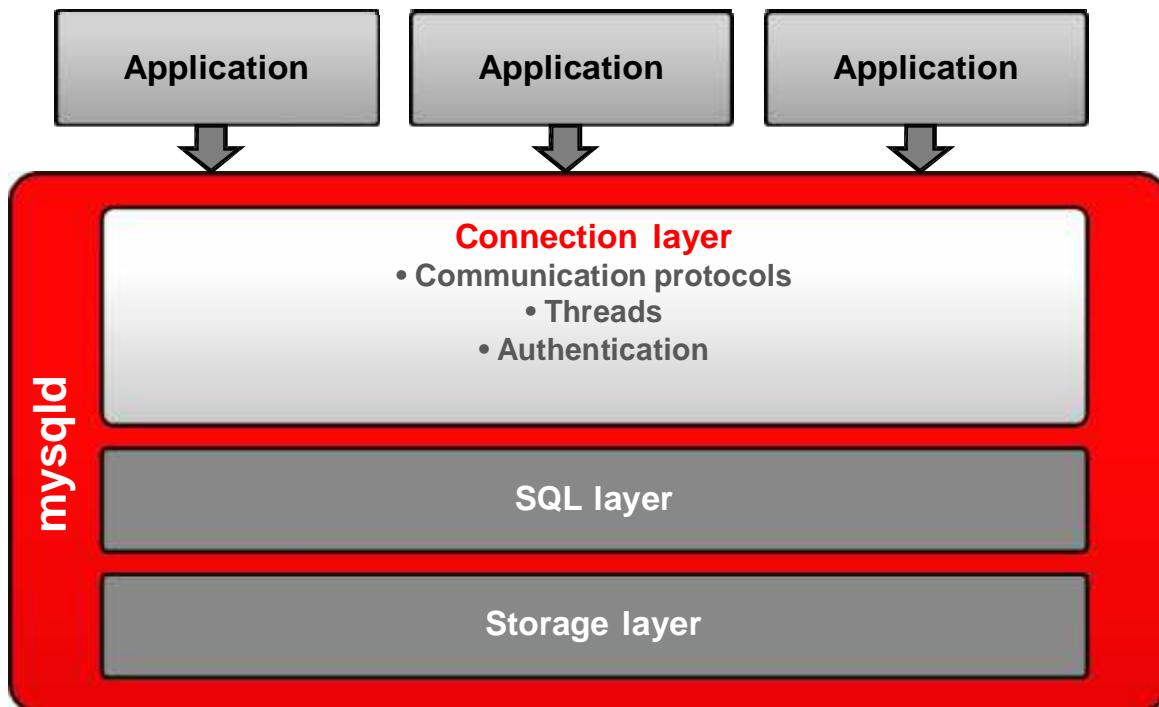
# Server Process



The `mysqld` (server program) process can be sliced into the following three layers:

- **Connection layer**: Handles connections. This layer exists on all server software (Web/mail/LDAP server).
- **SQL layer**: Processes SQL queries that are sent by connected applications
- **Storage layer**: Handles data storage. Data can be stored in different formats and structures on different physical media.

# Connection Layer



The connection layer accepts connections from applications over several communication protocols:

- TCP/IP
- UNIX sockets
- Shared memory
- Named pipes

TCP/IP works across the network. The other protocols listed above support only local connections when the client and server are running on the same machine. This layer maintains one thread per connection. This thread handles query execution. Before a connection can begin sending SQL queries, the connection is authenticated by verification of username + password + client host.

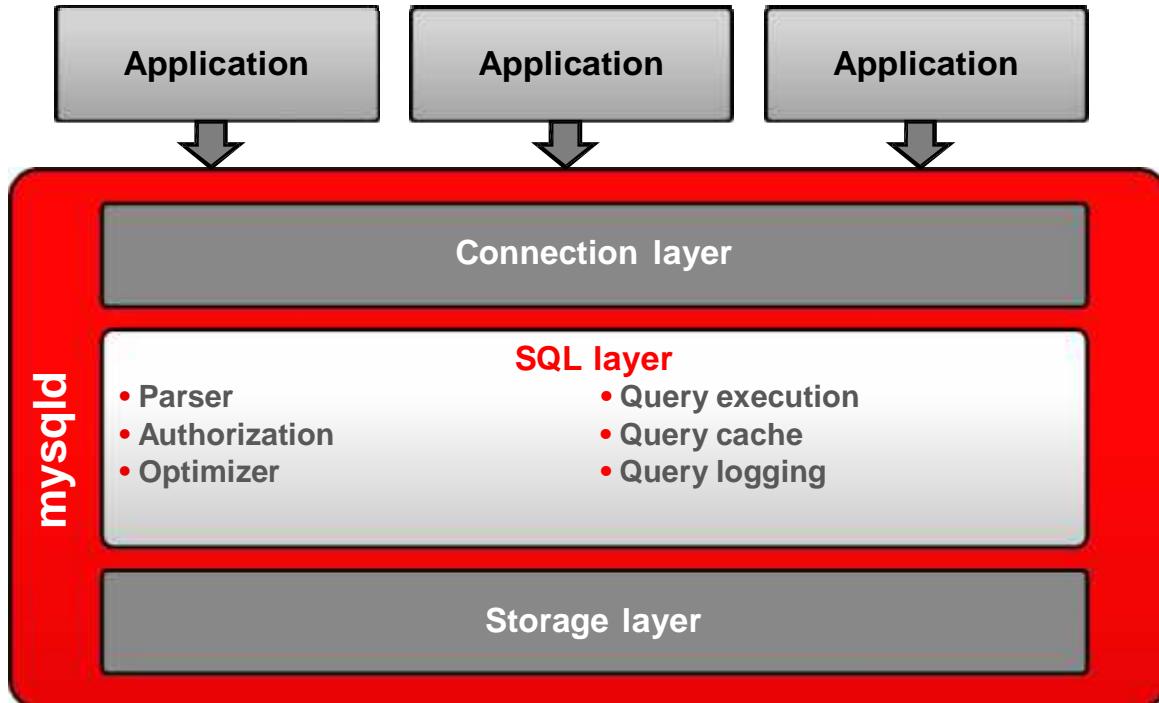
MySQL uses DNS (Domain Naming System) to resolve the names of hosts that connect using TCP/IP protocol, storing them in a host cache. For large networks that exhibit performance problems during name resolution, disable DNS with the `--skip-name-resolve` option, or increase the value of the `--host-cache-size` option.

# Communication Protocols

| Protocol         | Types of Connections | Supported Operating Systems |
|------------------|----------------------|-----------------------------|
| TCP/IP           | Local, remote        | All                         |
| UNIX socket file | Local only           | UNIX only                   |
| Shared memory    | Local only           | Windows only                |
| Named pipes      | Local only           | Windows only                |

- Protocols are implemented in the client libraries and drivers.
  - The speed of a connection protocol varies with the local settings.
- 
- **TCP/IP (Transmission Control Protocol/Internet Protocol):** The suite of communication protocols used to connect hosts on the Internet. In the Linux operating system, TCP/IP is built-in and is used by the Internet, making it the standard for transmitting data over networks. This is the best connection type for Windows.
  - **UNIX socket:** A form of inter-process communication used to form one end of a bidirectional communication link between processes on the same machine. A socket requires a physical file on the local system. This is the best connection type for Linux.
  - **Shared memory:** An efficient means of passing data between programs. One program creates a memory portion that other processes (if permitted) can access. This Windows explicit “passive” mode works only within a single (Windows) machine. Shared memory is disabled by default. To enable shared-memory connections, you must start the server with the --shared-memory option.
  - **Named pipes:** The use of named pipes is biased toward client/server communication, where they work much like sockets. Named pipes support read/write operations, along with an explicit “passive” mode for server applications. This protocol works only within a single (Windows) machine. Named pipes are disabled by default. To enable named-pipe connections, you must start the server with the --enable-named-pipe option.

## SQL Layer

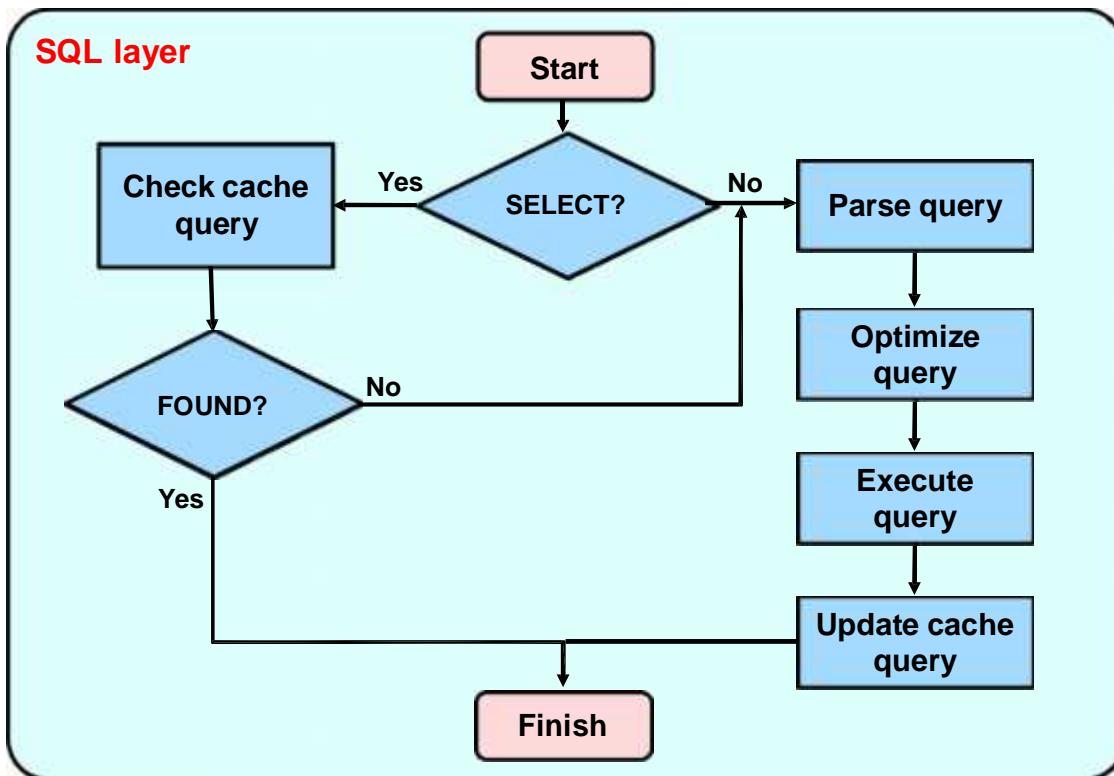


After a connection is established, the following processes are handled by the MySQL server:

- **Authorization and parser:** The parser validates the correct syntax, and then authorization verifies that the connected user is allowed to run a particular query.
- **Optimizer:** Creates the execution plan for each query, which is a step-by-step instruction set on how to execute the query in the most optimal way. Determining which indexes are to be used, and in which order to process the tables, is the most important part of this step.
- **Query execution:** Fulfills the execution plan for each query
- **Query cache:** Optionally configurable query cache that can be used to memorize (and immediately return) executed queries and results
- **Query logging:** Can be enabled to track executed queries

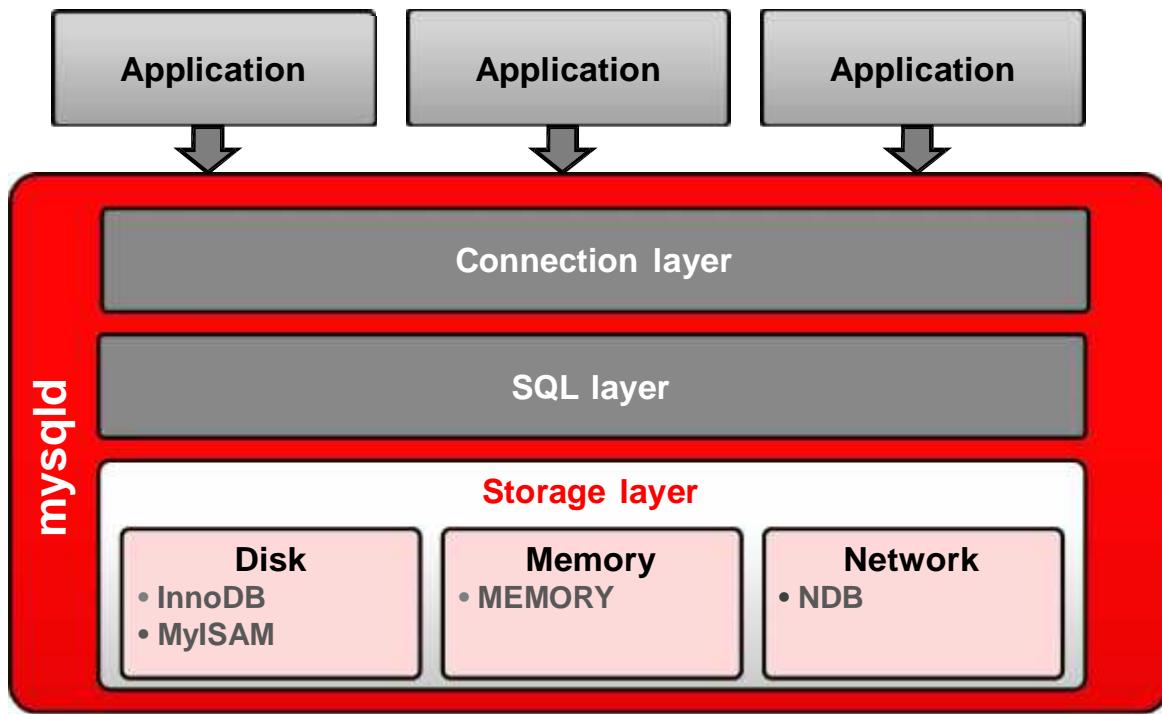
**Note:** The diagram in the slide shows how the MySQL server processes SQL statements.

# SQL Statement Processing



SQL statements are processed in the order and relationship described in the slide.

## Storage Layer



With MySQL, you can use different types of storage called “storage engines.” Data can be stored on disk, memory, and network. Each table in a database can use any of the available storage engines. *Disk* storage is cheap and persistent, whereas *memory* is much faster.

**InnoDB** is the default storage engine. It provides transactions, full-text indexing, and foreign key constraints, and is thus useful for a wide mix of queries. It is multipurpose and supports read-intensive, read/write, and transactional workloads.

Other storage engines include:

- **MyISAM:** Useful for mostly read and little update of data
- **MEMORY:** Stores all data in memory
- **NDB:** Used by MySQL Cluster to provide redundant scalable topology for highly available data

**Note:** Storage engines extend beyond just the storage layer, and consist of more than just storage. They also include other structures and implementation mechanisms.

# Storage Engine: Overview

Storage engines are server components that act as handlers for different table types.

- Storage engines are used to:
  - Store data
  - Retrieve data
  - Find data through an index
- Two-tier processing
  - Upper tier includes SQL parser and optimizer.
  - Lower tier consists of a set of storage engines.
- SQL tier is not dependent on the storage engine:
  - The engine does not affect SQL processing.
  - There are some exceptions.

A client retrieves data from tables or changes data in tables by sending requests to the server in the form of SQL statements. The server executes each statement by using the two-tier processing model.

Clients normally do not need to be concerned about which engines are involved in processing SQL statements. Clients can access and manipulate tables by using statements that are the same no matter which engine manages them. Exceptions to this engine-independence of SQL statements include the following:

- `CREATE TABLE` has an `ENGINE` option that specifies which storage engine to use on a per-table basis.
- `ALTER TABLE` has an `ENGINE` option that enables the conversion of a table to use a different storage engine.
- Some index types are available only for particular storage engines. For example, only the InnoDB and MyISAM engines support full-text indexes.
- `COMMIT` and `ROLLBACK` operations affect only tables that are managed by transactional storage engines such as InnoDB and NDB.

# Features Dependent on Storage Engine

- Storage medium
- Transactional capabilities
- Locking
- Backup and recovery
- Optimization
- Special features
  - Full-text search
  - Referential integrity
  - Spatial data handling

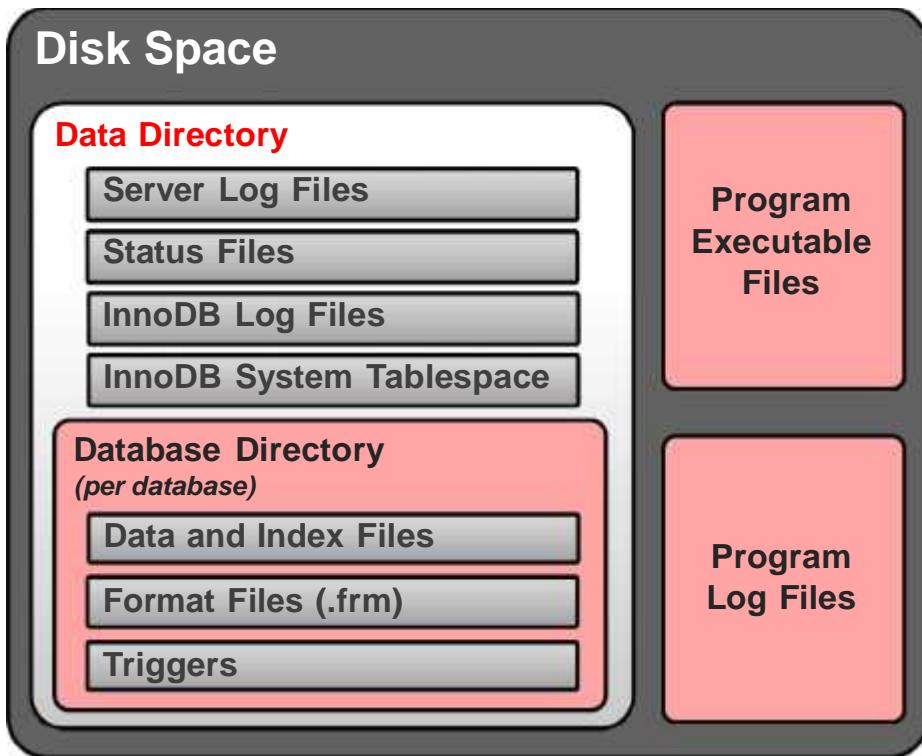
The following properties are dependent on the storage engine:

- **Storage medium:** A table storage engine can store data on disk, in memory, or over a network.
- **Transactional capabilities:** Some storage engines support full ACID transactional capability, while others may have no transactional support.  
**Note:** ACID is discussed in the lesson titled “Transactions and Locking.”
- **Locking:** Storage engines may use different locking granularity (such as table-level or row-level locks) and mechanisms to provide consistency with concurrent transactions.
- **Backup and recovery:** May be affected by how the storage engine stores and operates the data
- **Optimization:** Different indexing implementations may affect optimization. Storage engines use internal caches, buffers, and memory in different ways to optimize performance.
- **Special features:** Certain engine types have features that provide full-text search, referential integrity, and the ability to handle spatial data.

The optimizer may need to make different choices depending on the storage engine, but this is all handled through a standardized interface (API) that each storage engine supports.

**Note:** More information about storage engines and related concepts is provided later in this course.

# How MySQL Uses Disk Space



Program files are stored under server installation directories, along with the data directory. The program executable and log files are created by the execution of the various client, administrative, and utility programs. The primary use of disk space is the data directory.

- **Server log files** and **status files** contain information about statements that the server processes. Logs are used for troubleshooting, monitoring, replication, and recovery.
- **InnoDB log files** for all databases reside at the data directory level.
- **InnoDB System Tablespace** contains the data dictionary, undo log, and buffers.
- Each database has a single directory under the data directory, regardless of what types of tables are created in the database. The database directories store the following:
  - **Data files:** Storage engine-specific data files. These files can also include metadata or index information, depending on the storage engine used.
  - **Format files (.frm):** Contain a description of each table and/or view structure, located in the corresponding database directory
  - **Triggers:** Named database objects that are associated with a table and are activated when a particular event occurs for the table
- The location of the data directory depends on the configuration, operating system, installation package, and distribution. A typical location is `/var/lib/mysql`.

- MySQL stores the system database (`mysql`) on disk. `mysql` contains information such as users, privileges, plugins, help lists, events, time-zone implementations, and stored routines.

**Note:** More detailed information about the installation directories is provided in later lessons.

# How MySQL Uses Memory

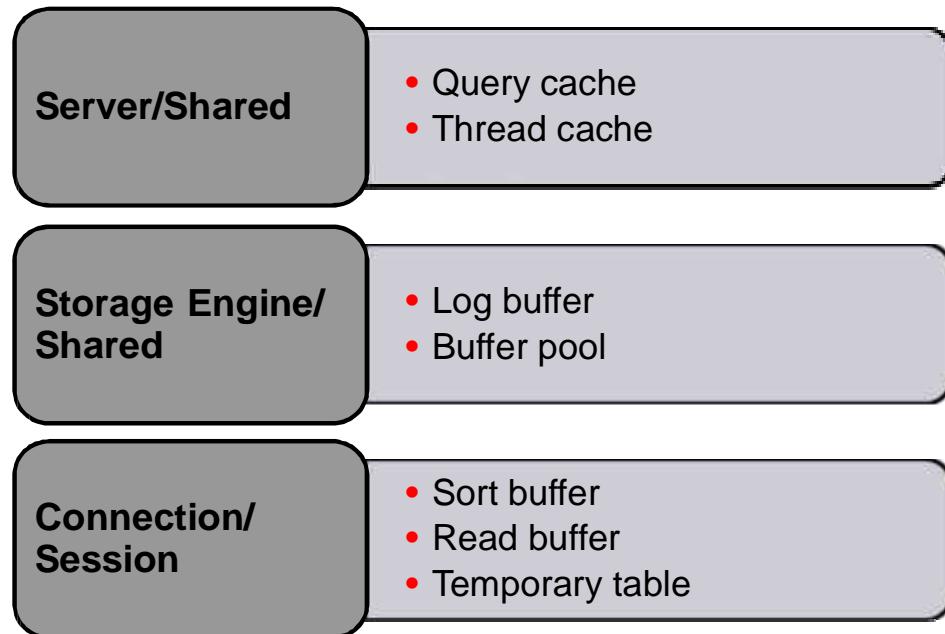
- Global
  - Allocated once
  - Shared by the server process and its threads
- Session
  - Allocated for each thread
  - Dynamically allocated and deallocated
  - Used for handling query results
  - Buffer sizes per session

Memory allocation can be divided into the following two categories:

- **Global (*per-instance* memory):** Allocated once when the server starts and freed when the server shuts down. This memory is shared across all sessions.  
When all the physical memory has been used up, the operating system starts swapping. This has an adverse effect on MySQL server performance and can cause the server to crash.
- **Session (*per-session* memory):** Dynamically allocated per session (sometimes referred to as *thread*). This memory can be freed when the session ends or is no longer needed.  
This memory is mostly used for handling query results. The sizes of the buffers used are per connection. For instance, a `read_buffer` of 10 MB with 100 connections means that there could be a total of  $100 \times 10$  MB used for all read buffers simultaneously.

# Memory Structures

Server allocates memory in three different categories:



The server allocates memory for many kinds of data as it runs.

- **Thread cache**: Threads are used in MySQL (and other programs) to split the execution of the application into two or more simultaneous running tasks. An individual thread is created for each client that connects to the MySQL server to handle that connection.
- **Buffers and caches**: Buffers and caches provide a data management subsystem and support fast-access items such as grant table buffers, storage engine buffers such as InnoDB's log buffers, and table open caches that hold descriptors for open tables. A query cache is also used to speed up the processing of queries that are issued repeatedly.

If you use the MEMORY storage engine, MySQL uses the main memory as principal data store. Other storage engines may also use main memory for data storage, but MEMORY is unique for being designed not to store data on disk.

## **Connection/Session**

- **Internal temporary tables:** In some cases of query execution, MySQL creates a temporary table to resolve the query. The temporary table can be created in memory or on disk depending on its size or contents or on the query syntax.
- **Client-specific buffers:** Are specifically designed to support the individual clients that are connected. Examples of the buffers include:
  - Communications buffer for exchanging information
  - Table read buffers (including buffers that support joins)
  - Sort operations

# MySQL Plugin Interface

- Daemon plugins, run by the server
- Plugin API allows loading and unloading of server components.
  - Supports dynamic loading, without restarting server
  - Supports full-text parser
  - Supports different authentication methods
  - Supports storage engines installed as plugins
  - Supports `INFORMATION_SCHEMA` plugins
- Requires the `PLUGINS` table in the `mysql` database
- MySQL supports both client and server plugins.

Currently, the plugin API supports:

- Full-text parser plugins that can be used to replace or augment the built-in full-text parser. For example, a plugin can parse text into words by using rules that differ from those used by the built-in parser. This is useful to parse text with characteristics different from those expected by the built-in parser.
- Storage engines that provide low-level storage, retrieval, and data indexing to the server
- Information schema plugins. An information schema plugin appears as a table in the MySQL `INFORMATION_SCHEMA` database. The `INFORMATION_SCHEMA` database is discussed later in more detail.
- A Daemon plugin starts a background process that runs within the server (for example, to perform heartbeat processing at regular intervals).

The plugin interface requires the `PLUGINS` table in the `mysql` database. This table is created as part of the MySQL installation process.

## **Summary**

In this lesson, you should have learned how to:

- Describe the MySQL client/server model
- Understand communication protocols
- Understand how the server supports storage engines
- Explain the basics of how MySQL uses memory and disk space

## Chapter 2

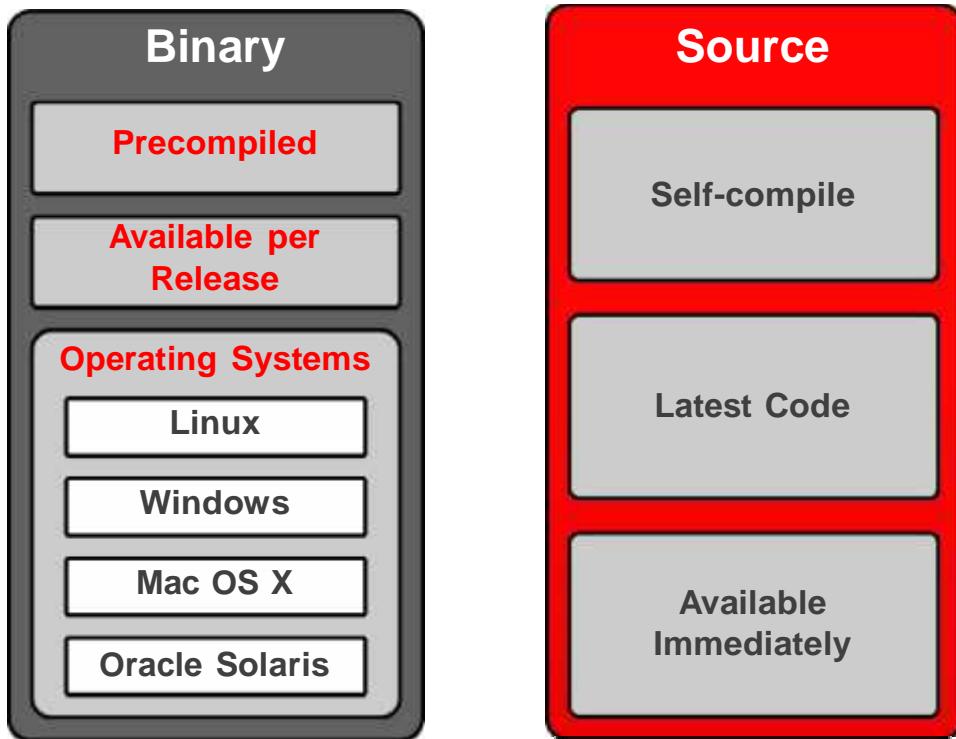
### Server Configuration and Client Tools

## **Objectives**

After completing this lesson, you should be able to:

- Choose the proper type of MySQL server software distribution
- Install the MySQL server
- Describe the MySQL server post-installation file structure
- Start and stop the MySQL server
- Upgrade MySQL
- Run multiple MySQL servers on a single host

# MySQL Server Distributions



MySQL is available for several operating systems, including Linux, Windows, Mac OS X, and Oracle Solaris. This course covers only Linux and Windows.

MySQL is available as both binary and source distributions:

- **Binary distributions:** Are precompiled, ready-to-run programs that are available for Enterprise and Community MySQL Server versions. These binaries are the official Oracle-tested versions.
- **Source distributions:** Are not guaranteed to be consistent with commercial code updates, nor do they include Oracle support

The slide shows only some of the available operating systems.

# MySQL Binary Distributions

| Binary Distributions |            |
|----------------------|------------|
| Linux                | Windows    |
| RPM Files            | Complete   |
| TAR Files            | No-install |

Advantages of using binary distributions:

- Created by experienced MySQL staff
- Good configuration options for improved performance
- High-quality commercial compilers
- Extensive libraries provided
- Include tested bug fixes, third-party patches, and features

## Binary for Linux

- **RPM** files are available for RPM-based Linux distributions, such as Oracle Linux. These files are installed by using the `rpm` program, or by using a package manager such as `yum`. The installation layout for each RPM is given by a specification file contained within the RPM file itself. (Use `rpm -qpl <rpm_file>` to determine where the contents of an RPM file are located upon installation.)

**Note:** See <http://rpm.org/> for more information about RPM files.

- **TAR** files are available for many varieties of Linux and UNIX-like systems. To install this kind of distribution, unpack it using the `.tar` program in the installation directory.
- For more information about specific packages available for Linux distributions, see <http://dev.mysql.com/downloads/mysql/>.

## Binary for Windows

- **Complete distribution:** Contains all files for a MySQL installation, as well as the configuration wizard
- **No-install distribution:** A `.zip` archive for which no installation or configuration wizard is used. You simply unpack it and move it to the desired installation location.

Binary distributions are also available for several other operating systems (including Oracle Solaris) as compressed files.

# MySQL Source Distributions

- Are used when:
  - No binary exists for your particular platform
  - You need to enable a source-only feature
  - You need to disable a feature that is not needed
  - You need to access the most current code
- Require you to compile the source code
- Are provided as compressed `.tar` archives

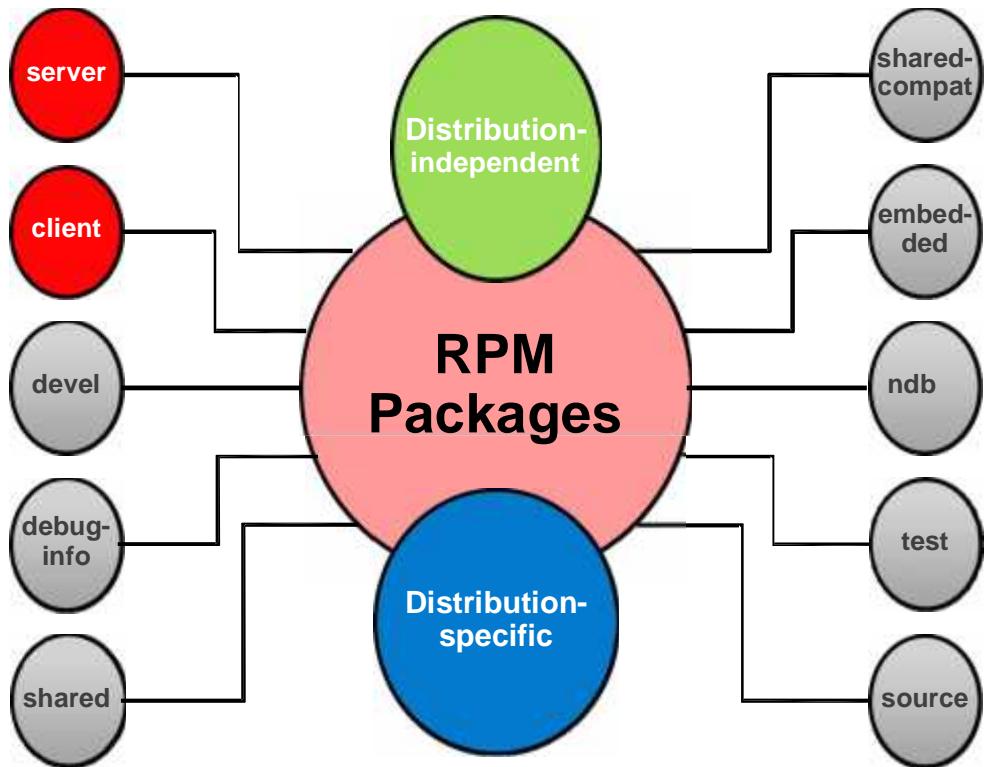
You should compile MySQL from source code if you require a feature that might not be available in a precompiled distribution (such as full debugging support). To produce a server that uses less memory when it runs, you may need to disable a feature that is not needed. For example, you may need to disable optional storage engines, or compile only those character sets that are really needed.

Binary distributions are available only for released versions and not for the very latest development source code.

A source distribution can be installed at any desired location. The default Linux installation location is `/usr/local/mysql`.

You can find the basic commands to install a MySQL source distribution in the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/source-installation.html>.

# MySQL RPM Installation Files for Linux



The recommended way to install MySQL on RPM-based Linux distributions is by using the RPM packages, in the form of .rpm files.

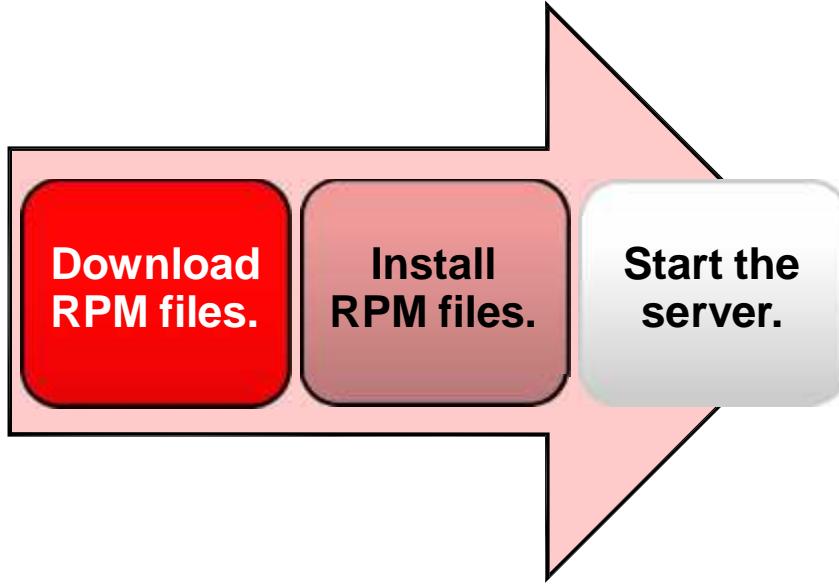
RPM files are provided for many platforms, such as Oracle Linux, SuSE, and other Linux versions. Many more distributions are available for other types of \*nix systems, including Oracle Solaris, FreeBSD, and Mac OS X.

Oracle provides two types of MySQL RPMs:

- **Distribution-independent:** The RPMs that MySQL provides to the community, which should work on all versions of Linux that support RPM packages and use glibc 2.3
- **Distribution-specific:** Intended for a targeted Linux platform

An RPM installation for MySQL is typically split into different packages. For a standard installation, you must install at least the server and client programs. The other packages are not required for a standard installation. For specific details about each of the RPM package files, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/linux-installation-rpm.html>.

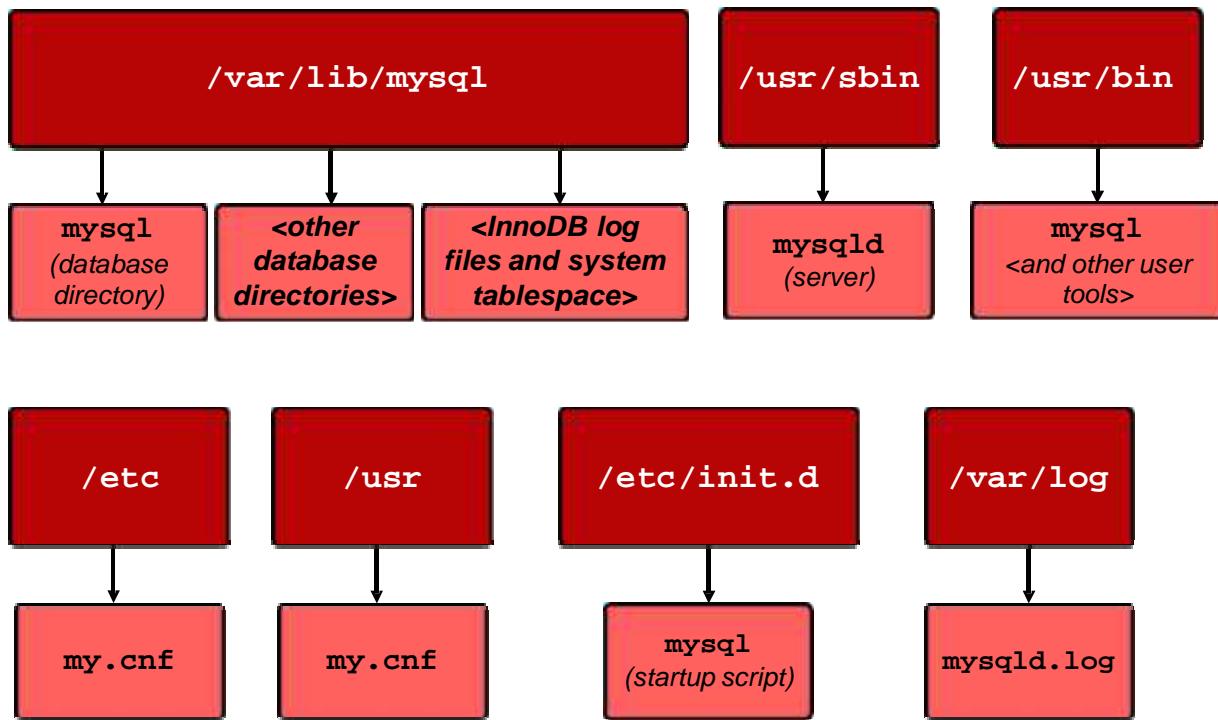
# Linux MySQL RPM Installation Process



To install the RPM files and start the server:

1. Download the appropriate RPM files (for your Linux version) from the MySQL downloads website.
2. Install all the downloaded RPM files.
  - a. Execute the `rpm -i <rpm_filename>` command for each RPM file.
  - b. The installation performs the following tasks automatically as it runs:
    - Extracts RPM files to their default locations
    - Registers a startup script named `mysql` in the `/etc/init.d` directory
    - Executes `mysql_install_db`, a script that creates the system databases and default `my.cnf` file, sets up a random password for the `root` accounts, and saves that password in the installing user's home directory in a file called `.mysql_secret`
    - Sets up a login account with user and group names for `mysql` (for administering and running the server)
3. Start the MySQL server.
  - a. Execute the `/etc/init.d/mysql start` command to start the MySQL server.

# Linux MySQL Server Installation Directories



When the RPM files are unpacked, many files are automatically extracted and placed into different locations. The slide shows the most common default directory locations and files.

## Data Directory

`/var/lib/mysql` is where the server stores databases. This directory is preconfigured and ready to use. For example, this directory includes a `mysql` subdirectory (contains the grant tables) and a `test` directory for the `test` database (can be used for testing purposes). The InnoDB log files and system tablespace are in this directory.

## Base Directory

`/usr` is the *base directory*. It contains program files and a `my.cnf` file. When you use `mysqld_safe` to launch the MySQL Server process (`mysqld`), it uses the `my.cnf` file stored in this location.

`/usr/bin` contains client programs and scripts such as `mysql`, `my_print_defaults`, `mysqladmin`, `mysqlcheck`, `mysqld_safe`, and `innochecksum`.

## Other Directories

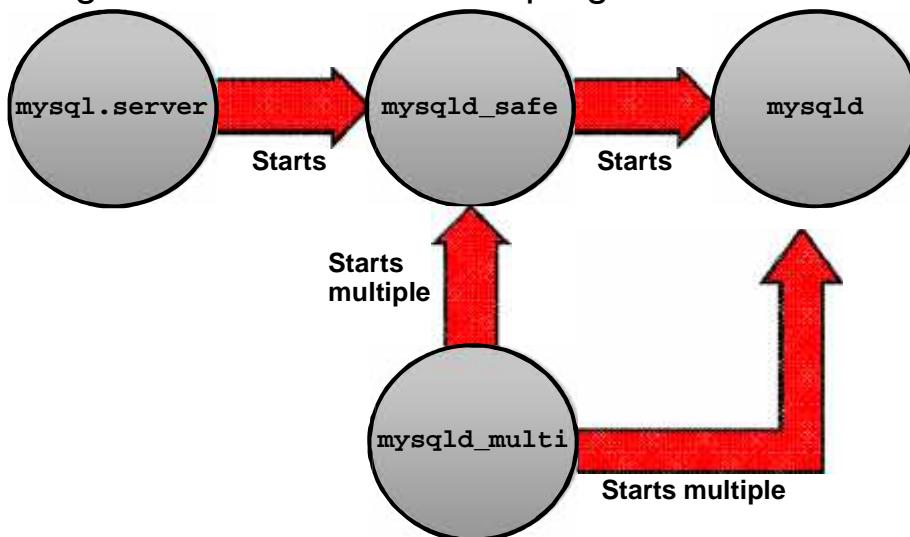
`/etc` and `/var/log` are standard Linux directories for configuration files and log files. The `/etc/my.cnf` file is read by the MySQL Server process (`mysqld`). For more information about the use of configuration options files see the lesson titled “Server Configuration”.

# Starting the MySQL Server on Linux

- The server can be started from a command line:

```
service mysql start
```

- Starting the server follows this progression:



The server can be started on Linux using several methods:

- mysql.server**: Used as a wrapper around `mysqld_safe` for systems such as Linux and Oracle Solaris that are using System V run-level directories
- mysqld\_safe**: Sets up the error log and then launches `mysqld` and monitors it. If `mysqld` terminates abnormally, `mysqld_safe` restarts it. If the server does not start properly, look in the error log.
- mysqld**: Invokes the server manually to debug the MySQL server. The error messages go to the terminal by default rather than to the error log.
- mysqld\_multi**: Perl script that is intended to simplify the management of multiple servers on a single host. It can start or stop servers, or it can report whether servers are running.

Install the correct script to have the server run automatically at startup:

- On BSD-style Linux systems, it is most common to invoke `mysqld_safe` from one of the system startup scripts, such as the `rc.local` script in the `/etc` directory.
- Linux and UNIX System V variants with run-level directories under `/etc/init.d` use the `mysql.server` script. Prebuilt Linux binary packages install `mysql.server` under the name `mysql` for the appropriate run levels. Invoke it manually with an argument of `start` or `stop`, either directly or by using the `service` command.

# Stopping the MySQL Server on Linux

- Methods for stopping the server:

| Script                    | Method Description   |
|---------------------------|--|
| <code>mysqladmin</code>   | Connects to the server as a client to shut down the server (local or remote)     |
| <code>mysql.server</code> | Stops and/or shuts down the local server   |
| <code>mysqld_multi</code> | Invokes <code>mysqladmin</code> to stop and/or shut down servers that it manages |

- Stop the server from command line:

```
service mysql stop
```

- Check whether the server is running or not:

```
service mysql status
```

To stop the server manually, use one of the following techniques:

- `mysqladmin`: Has a shutdown command. It connects to the server as a client and can shut down local or remote servers.
- `mysql.server`: Stops and/or shuts down the local server when invoked with an argument of stop
- `mysqld_multi`: Stops and/or shuts down any of the servers that it manages. It does so by invoking `mysqladmin`

`mysqld_safe` has no server shutdown capability. You can use `mysqladmin shutdown` instead. Note that if you forcibly terminate `mysqld` by using the `kill -9` command to send it a signal, then `mysqld_safe` detects that `mysqld` terminated abnormally and restarts it. You can work around this by killing `mysqld_safe` first and then killing `mysqld`, but it is better to use `mysqladmin shutdown`, which initiates a normal (clean) server shutdown.

# Improving Installation Security

- Invoke the security program:

```
mysql_secure_installation
```

- Prompts for actions to perform
- Applicable to UNIX and Linux operating systems only
- Security improvements
  - Sets or changes password for `root` accounts
  - Removes remote `root` accounts
  - Removes anonymous accounts
  - Removes the `test` database

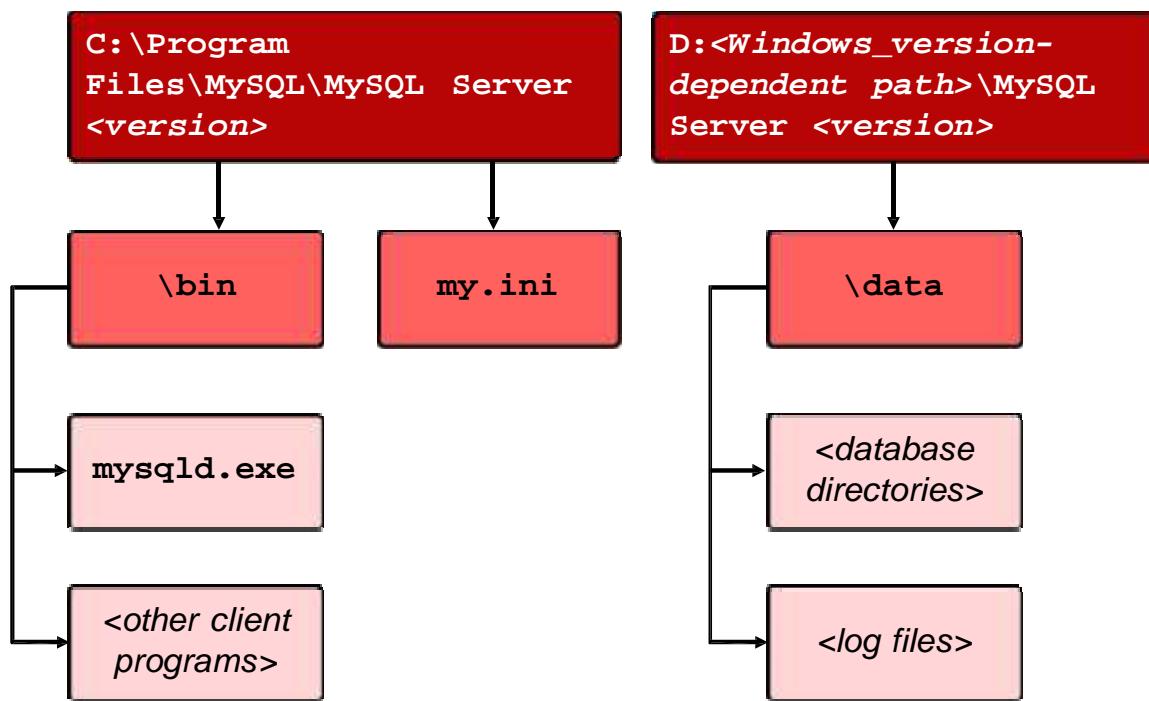
When you install MySQL from an RPM package, it sets up a random password for the root account and saves that password to the `.mysql_secret` file in the installing user's home directory. For all other installations, the initial passwords are blank.

As a result, you should set up passwords as soon as the server has been started.

Invoke `mysql_secure_installation` without arguments, which prompts you to determine which actions to perform.

For a full discussion of the postinstallation procedure for UNIX operating systems, see <http://dev.mysql.com/doc/mysql/en/unix-postinstallation.html>.

# Windows MySQL Server Installation Directory



By default, MySQL 5.6 is installed at the directory path `C:\\Program Files\\MySQL\\MySQL 5.6 Server`.

## Bin Directory

`\bin` contains the MySQL server and client programs. Windows MySQL distributions include multiple programs, which can be found in the `bin` directory:

- `mysqld.exe`: Standard server
- Other client programs, such as `mysqladmin.exe` (see the description in the Linux section)

## Data Directory

`\data` is where the server stores databases and log files. This directory is preconfigured and ready to use. For example, this directory includes a `mysql` subdirectory (contains the grant tables) and the `test` subdirectory for the `test` database (can be used for testing purposes).

## Configuration File

The `my.ini` configuration option file specifies where the installation directory is located, as well as other optional settings.

# Running MySQL on Windows

| Methods for Running MySQL on Windows   |   |
|--|---|
| <b>Server</b> <ul style="list-style-type: none"><li>• Start manually.</li><li>• Stop manually.</li></ul> | <b>Service</b> <ul style="list-style-type: none"><li>• Install manually.</li><li>• Start/stop manually or automatically.</li><li>• Use GUI.</li></ul> |

Run a Windows MySQL server manually from the command line of a console window by using the `mysqld` command (with extra options, if needed). You stop the server by using the `mysqladmin shutdown` command.

To run the MySQL server so that Windows itself starts and stops the server when Windows starts and stops, install the server as a Windows service. To do this, invoke the server from the command line using the `mysqld --install` command (with extra options if needed).

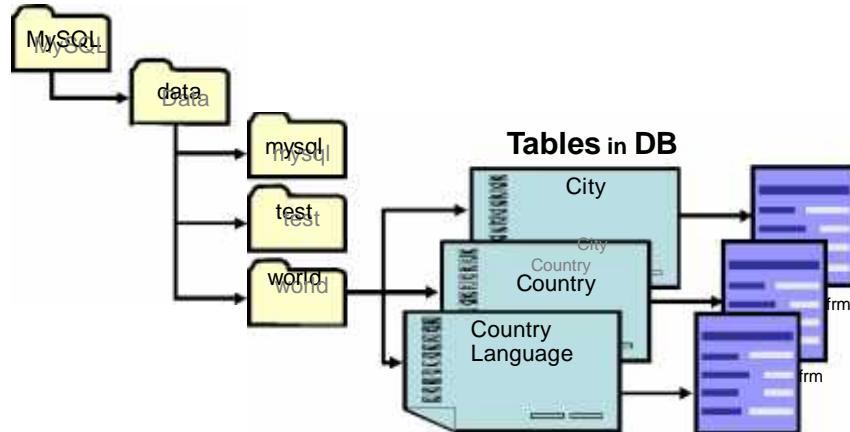
You can also start and stop the service manually from the command line by using the `net start MySQL` and `net stop MySQL` commands.

To start and stop the service by using the Windows Services GUI, select the MySQL service in the Administrative Tools and then click the Start or Stop link. You can configure manual or automatic startup in the Services GUI.

To remove the service after it has been stopped, use `mysqld` with the `--remove` option.

# Data Directory

- Example



- Implications of database directory mapping:
  - Database and table names are case-sensitive (according to OS setup).
  - Databases can be mounted on separate physical devices to improve performance.

The general structure of the data directory (regardless of OS) is shown in this slide. Parts of the structure are specific to the storage engine. However, every table has a `*.frm` file (including views) regardless of the storage engine being used. The MySQL server changes its current working directory to its `data` directory when it begins executing.

You must make sure that the MySQL server has the proper access rights to create files in the `data` directory. The server must have access rights in any directory where it needs to create data files or log files.

The MySQL server maps each database to a directory under the MySQL `data` directory, and by default it maps tables in a database to file names in the database directory. This has the following implications:

- Database and table names are case-sensitive in the MySQL server only on operating systems that have case-sensitive file names (such as most Linux systems).
- You can split the disk usage by moving the data directory, databases, and/or single tables (depending on the storage engine options) to different physical locations, which can improve performance.

# MySQL Server Releases

- Oracle frequently releases new MySQL versions:
  - New features
  - Bug fixes
- You can check implications before upgrading.
- Upgrade procedure:



- The upgrade might require reconfiguration.
- You can evaluate the impact of the upgrade in a test environment.

Prior to performing an upgrade to a newer version of MySQL, check the implications and possible difficulties. You should also check the *MySQL Reference Manual* before upgrading:

- In the section on upgrading, always read the notes pertaining to the type of upgrade that is being performed. Follow the recommended procedures.
- In the change note sections for new versions, check for all the changes that have taken place between the current version and the version to be installed. Note any changes that are not backward-compatible with the current version.

RPM and source upgrades do not usually require reconfiguration because they tend to use the same installation directory location regardless of the MySQL version. However, if you upgrade using generic Linux binaries, you may choose to create a new version-specific directory to contain the upgraded release. Also, Windows installers create version-specific folders under Program Files. In these cases, you may need to do some reconfiguration.

Set up a symbolic link that points to the old installation directory so that it can be easily deleted and re-created to point to the new installation directory. Subsequent references to the symbolic link access the new installation. If your installation was originally produced by installing multiple RPM packages, it is best to upgrade all the packages rather than just some of them. For example, if you previously installed the server and client RPMs, do not upgrade only the server RPM.

# Checking Upgraded Tables

1. Make sure that the MySQL server is running.
2. Execute the following for every upgrade:

```
mysql_upgrade [options]
```

3. Check table status.
4. Check and repair tables:

```
mysqlcheck [options]
```

5. Upgrade system tables.
6. Mark upgraded tables with the current version.
7. Save the new version in a file:

```
mysql_upgrade.info
```

## Execute Upgrade Program

- **mysql\_upgrade**: Reads options from the command line and from the [mysql\_upgrade] and [client] option file groups

## Check and Repair Tables

Run `mysql_upgrade` to:

- Check all tables in your databases for incompatibilities with current versions of the MySQL server
- Repair any problems found in tables with possible incompatibilities
- Upgrade system tables to add any new privileges or capabilities that are available in the new version
- Mark all checked and repaired tables with the current MySQL version number

To check and repair tables and to upgrade the system tables, `mysql_upgrade` executes the `mysqlcheck` command.

For more information about usage and options for the `mysql_upgrade` program, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/mysql-upgrade.html>.

# Employing Multiple Servers

- Useful for many administrative purposes, such as:
  - Testing a new release of MySQL
  - Partitioning client groups into different servers
- Servers must not interfere with each other.
- **mysqld\_multi** allows you to manage multiple similar servers with different settings.

Run multiple servers when you want to test a new release of MySQL on the same machine where the production server is running. Give each group its own designated root user who is not able to see databases that belong to other groups, as would be possible if all clients were to share the same server.

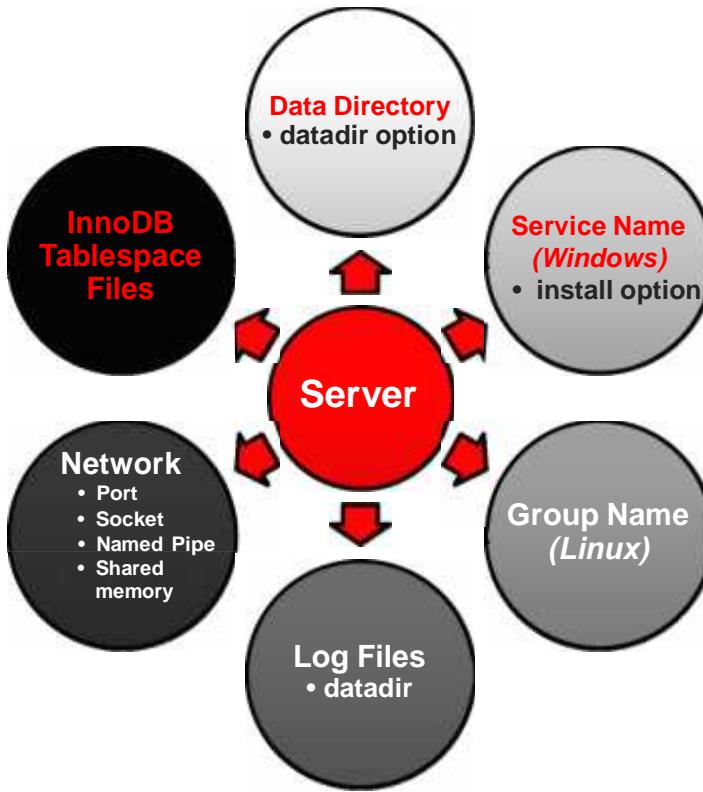
Do not allow any of the servers to share resources that must be used exclusively by a single server.

The `mysqld_multi` script is designed to manage several `mysqld` processes that listen for connections on different UNIX socket files and TCP/IP ports. It searches for groups named `[mysqldN]` in `my.cnf`, and it applies the settings for that `N` to the numbered instance.

For example, to start two `mysqld` instances that apply settings from `my.cnf` sections `[mysqld3]` and `[mysqld5]` respectively, run the following command:

```
shell> mysqld_multi start 3, 5
```

# Multiple Server Options

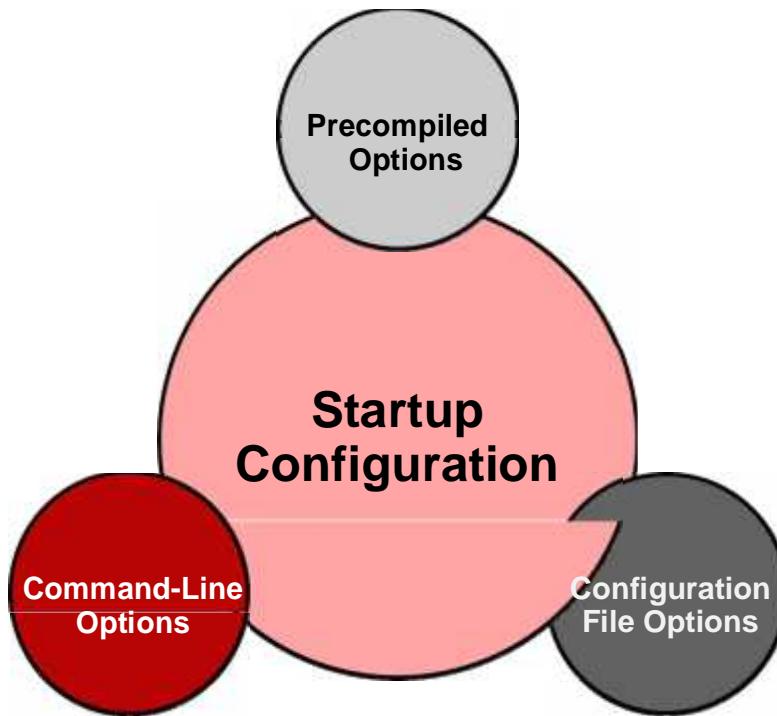


Invoke each MySQL server with `mysqld` or `mysqld_multi` and the corresponding options for each server function:

- **Data Directory:** Starts each server with a unique value for the `--datadir` option
- **Network:** Sets servers to use their own network interfaces by starting each server with a unique value for the `--port`, `--socket`, and `--shared-memory-basename` options
- **Group Name:** Each server group must have a unique `mysqldN` name on Linux or UNIX, when using `mysqld_multi`.
- **Log Files:** Each server must have its own log and PID files. Specify log and PID files with the `--log` and `--pid-file` options.
- **InnoDB Tablespace and Log Files:** Cannot be shared by multiple servers
- **Temporary Directory:** Having different temporary directories makes it obvious which server created a particular temporary file. Specify temporary directories with the `--tmpdir` option.
- **Windows Service Name:** Each `mysqld` Windows service must use a unique service name. Set the service names by using `--install`. When the servers start, they read options from the corresponding individual service groups in the standard option files.

Another approach is to specify the base directory for each installation with the `--basedir` option. This causes each instance to automatically use a different data directory and its own log and PID files because the default for each of those parameters is relative to the base directory.

# MySQL Configuration Options



You can specify startup options on the command line when you invoke the server (or client), or in an option file. MySQL client programs look for option files at startup and use appropriate options.

By default, the server uses precompiled values for its configuration variables when it runs. However, if the default values are not suitable for your environment, add runtime options to tell the server to use different values to:

- Specify the locations of important directories and files
- Control which log files the server writes
- Override the server's built-in values for performance-related variables (that is, to control the maximum number of simultaneous connections and the sizes of buffers and caches)
- Enable or disable precompiled storage engines at server startup

You can specify runtime options when the server is started (to change its configuration and behavior) by using command-line options or an option file, or by using a combination of both. Command-line options take precedence over any settings in an option file.

To find out what options your server supports, execute the following at a shell prompt:

```
mysqld --verbose --help
```

**Note:** The preceding command provides information. It does not start the MySQL server.

## Reasons to Use Option Files

- No need to specify options each time you start the server
- Lower chance for error when you enter options
- Opportunity to quickly review a single file to view the server configuration

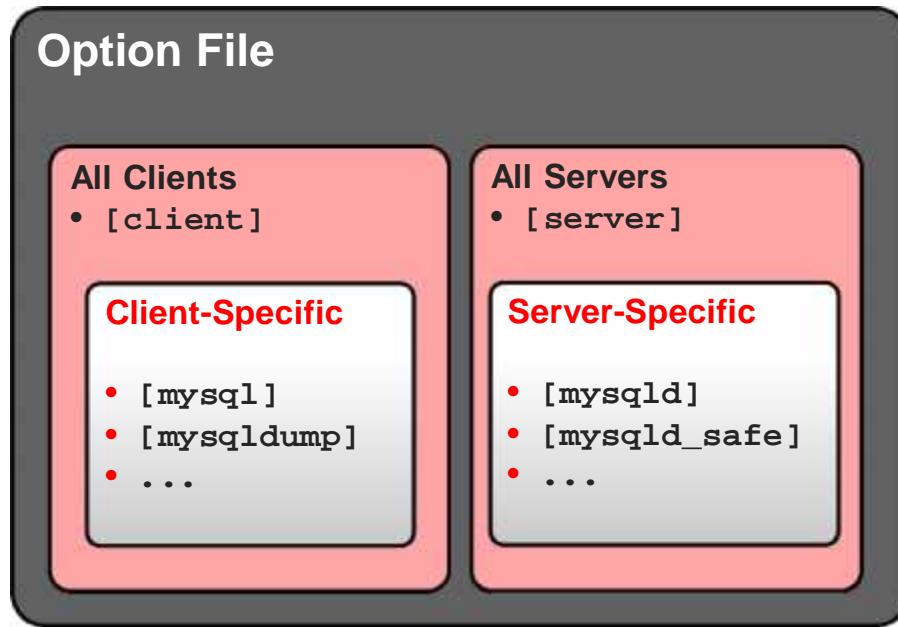
When you invoke the server from the command line, you can specify any of the server options listed with the `--help` option. However, it is more useful to list them in an option file, for several reasons:

- When you put options in a file, you do not need to specify them on the command line each time you start the server. This is more convenient and less error-prone for complex options, such as those used to configure the InnoDB tablespace.
- If a single option file contains all server options, you can see how the server has been configured at a glance.

MySQL programs can access options from multiple option files. Programs look for each of the standard option files and read any that exist. No error occurs if a given file is not found.

To use an option file, create it as a plain text file by using an editor. To create or modify an option file, you must have write permission for it. Client programs need only read access.

# Option File Groups



Options in option files are organized into groups, with each group preceded by a `[group-name]` line that names the group. Typically, the group name is the category or name of the program to which the group of options applies. Examples of groups include:

- **[client]**: Used for specifying options that apply to all client programs. A common use for the `[client]` group is to specify connection parameters, because typically connections are made to the same server no matter which client program is used.
- **[mysql]** and **[mysqldump]**: Used for specifying options that apply to `mysql` and `mysqldump` clients, respectively. Other client options can also be specified individually.
- **[server]**: Used for specifying options that apply to both the `mysqld` and `mysqld_safe` server programs
- **[mysqld]**, **[mysqld-5.6]**, and **[mysqld\_safe]**: Used for specifying options to different server versions or startup methods

## Writing an Option File

A brief example of groups in an option file:

```
[client]
host = myhost.example.com
compress

[mysql]
show-warnings
```

To create or modify an option file, the end user must have *write* permission for it. The server itself needs only *read* access; it reads option files but does not create or modify them.

To write an option in an option file:

- Use the long option format, as used on the command line, but omit the leading dashes.
- If an option takes a value, spaces are allowed around the equal ( = ) sign. This is not true for options specified on the command line.

In the example in the slide, note the following:

- **[client]**: Options in this group apply to *all* standard clients.
  - **host**: Specifies the server host name
  - **compress**: Directs the client/server protocol to use compression for traffic sent over the network.
- **[mysql]**: Options in this group apply *only* to the mysql client.
  - **show-warnings**: Tells MySQL to show any current warnings after each statement
- The mysql client uses options from *both* the [client] and [mysql] groups, so it would use all three options shown.

## Option File Locations

- MySQL server looks for files in standard locations.
- Standard files are different for Linux and Windows:
  - In Linux, use the **my.cnf** file.
  - In Windows, use the **my.ini** file.
- There is no single **my.cnf** or **my.ini** file. The server might read multiple **my.cnf** or **my.ini** files from different locations.
- You can view option file lookup locations, the order in which they are read, and groups with options:

```
shell> mysql --help
... Default options are read from the following files in
the given order:
/etc/my.cnf /etc/mysql/my.cnf /usr/etc/my.cnf ~/.my.cnf
The following groups are read: mysql client ...
```

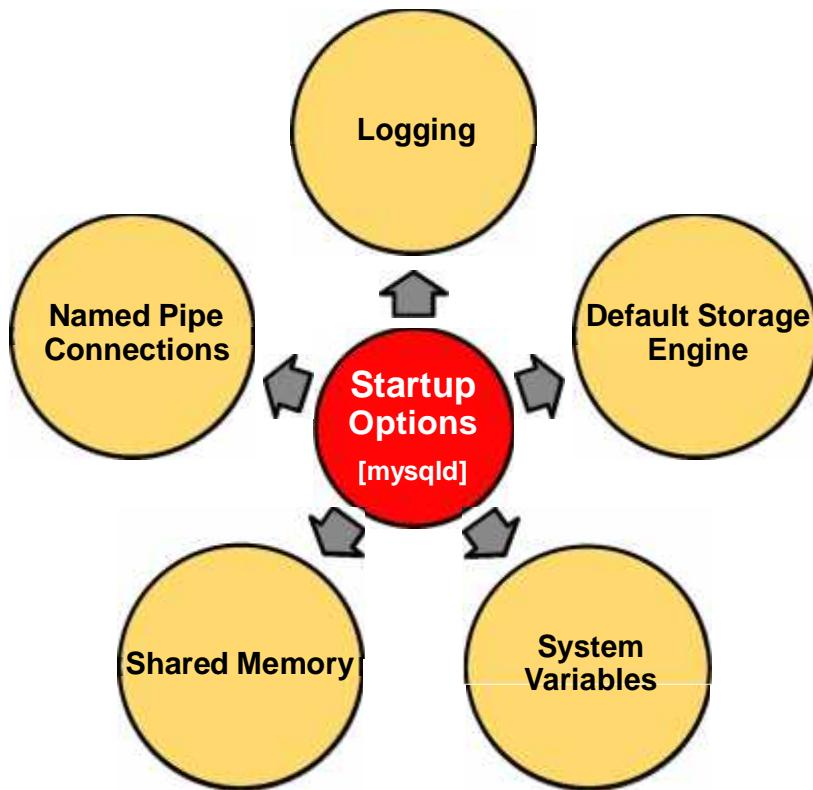
The standard option files are as follows:

- **Linux:** The file `/etc/my.cnf` serves as a global option file used by all users. You can create a user-specific option file named `.my.cnf` in the user's home directory. If the `MYSQL_HOME` environment variable is set, it searches for the `$MYSQL_HOME/my.cnf` file.
- **Windows:** Programs look for option files in the following order: `my.ini` and `my.cnf` in the Windows `C:\` directory, then the `C:\Windows` (or `C:\WinNT`) directory. However, because the Windows installation wizard places the configuration file in the directory `C:\Program Files\MySQL\MySQL Server <version number>`, the server also searches this directory in Windows.
- MySQL command-line programs search for option files in the MySQL installation directory.

To view the locations of options file and the order in which they are read, use the `mysql` command-line client with the `--help` option. The following command filters the output of the command:

```
mysqld --help --verbose 2> /dev/null | grep -A1 "Default options"
```

# Startup Options in an Option File



To specify server options in an option file, indicate the specific options under the `[mysqld]` or `[server]` groups.

- **Logging:** You can enable logging for your server by turning on the types of logs required. The following options turn on the general query log, the binary log, and the slow query log:  
`general_log`  
`log-bin`  
`slow_query_log`
- **Default Storage Engine:** You can specify a default storage engine different from InnoDB by using the `--default-storage-engine` option.
- **System Variables:** You can customize your server by setting server system variable values. For example, to increase the maximum allowed number of client connections and to increase the number of the InnoDB buffer pools from their defaults, set the following variables:  
`max_connections=200`  
`innodb_buffer_pool_instances=4`
- **Shared Memory:** Not enabled by default on Windows. You can turn on shared memory by using the `shared-memory` option.
- **Named Pipes:** To turn on named-pipe support, use the `enable-named-pipe` option.

## Sample Option Files

- The MySQL installation provides a sample configuration file called **my-default.cnf**.
  - The my-default.cnf file is not read by MySQL programs.
- The install process copies the my-default.cnf sample file to **/etc/my.cnf**.
  - If /etc/my.cnf already exists, it copies it to **/etc/my-new.cnf** instead.
- Specify new options in the file to replace, add, or ignore the standard options.
  - Must be the first option on the command line
    - defaults-file=<file\_name>**
    - defaults-extra-file=<file\_name>**
    - no-defaults**

- **Linux:** The my-default.cnf sample option file is in /usr/share/mysql for RPM installations or the share directory under the MySQL installation directory for TAR file installations.
- **Windows:** The my-default.ini sample option file and my.ini are located in the MySQL installation directory.

Before you change any of the default options, make sure that you fully understand the effects that the options have on server operation.

- **--defaults-file=<file\_name>**: Use the option file at the specified location.
- **--defaults-extra-file=<file\_name>**: Use an additional option file at the specified location.
- **--no-defaults**: Ignore all option files.

For example, to use only the /etc/my-opt.cnf file and ignore the standard option files, invoke the program like this: shell> mysql --defaults-file=/etc/my-opt.cnf

If an option is specified multiple times, either in the same option file or in multiple option files, the option value that occurs *last* takes precedence.

.

# Displaying Options from Option Files

Display options per group from the command line.

- Examples of [mysql] and [client] groups:

```
shell> my_print_defaults mysql client
--user=myusername
--password=secret
--host=localhost
--port=3306
--character-set-server=latin1
```

- Or (for the same output):

```
mysql --print-defaults mysql client
```

You can view which options are used by programs that read the specified option groups by executing the mysql client with the `--print-defaults` option or by using the `my_print_defaults` utility. The output consists of options, one per line, in the form that they would be specified on the command line. This output varies according to your option file settings.

`my_print_defaults` accepts the following options:

- `--help, -?:` Display a help message and exit.
- `--config-file=<file_name>, --defaults-file=<file_name>, -c <file_name>:` Read only the given option file.
- `--debug=<debug_options>, -# <debug_options>:` Write a debugging log.
- `--defaults-extra-file=<file_name>, --extra-file=<file_name>, -e <file_name>:` Read this option file after the global option file, but (on Linux) before the user option file.
- `--defaults-group-suffix=<suffix>, -g <suffix>:` Read groups with this suffix.
- `--no-defaults, -n:` Return an empty string.
- `--verbose, -v:` Verbose mode. Print more information about what the program does.
- `--version, -v:` Display version information and exit.

# Obscuring Authentication Options

Use `mysql_config_editor` to create encrypted option files.

- Store user, password, and host options in a dedicated option file:
  - `.mylogin.cnf` in the current user's home directory
  - To specify an alternative file name, set the `MYSQL_TEST_LOGIN_FILE` environment variable.
- The `.mylogin.cnf` file contains *login paths*.
  - They are similar to option groups.
  - Each login path contains authentication information for a single identity.
  - Clients refer to a login path with the `--login-path` command-line option:

```
mysql --login-path=admin
```

Specifying a password on the command line in the form `mysql -uroot -poracle` is not recommended. For convenience, you could put a password in a [client] option group, but the password is stored in plain text, easily visible to anyone with read access to the option file.

The `mysql_config_editor` utility enables you to store authentication credentials in an encrypted login file named `.mylogin.cnf`. The file location is the current user's home directory on Linux and UNIX, and the `%APPDATA%\MySQL` directory on Windows. The file can be read later by MySQL client programs to obtain authentication credentials for connecting to MySQL Server. The encryption method is reversible, so you should not assume the credentials are secure against anyone with read privileges to the file. Rather, the feature makes it easier for you to avoid using plaintext credentials.

The unencrypted format of the `.mylogin.cnf` login file consists of option groups, similar to other option files. Each option group in `.mylogin.cnf` is called a "login path," which is a group that permits only a limited set of options: host, user, and password. Think of a login path as a set of values that indicate the server host and the credentials for authenticating with the server. Here is an example:

```
[admin]
user = root
password = oracle
host = 127.0.0.1
```

# Login Paths

- To create a login path:

```
mysql_config_editor set  
    --login-path=<login-path> --user=<user>  
    --password --host=<hostname>
```

- To view a single login path in clear text:

```
mysql_config_editor print  
    --login-path=<login-path>
```

- To view all login paths in clear text:

```
mysql_config_editor print --all
```

- To remove a login path:

```
mysql_config_editor remove  
    --login-path=<login-path>
```

- The default login path name is `client`. It is read by all standard clients.

If you invoke `mysql_config_editor` without using the `--login-path` option, it uses the `[client]` login path. This login path is used by all standard clients by default.

For example, the following command creates a `[client]` login path used by all standard clients:

```
shell> mysql_config_editor set --user=root --password  
Enter password: oracle
```

Invoking a standard client with no further command-line arguments or option files causes it to read the `[client]` login path in the `.mylogin.cnf` file, along with `[client]` option groups in any option files. For example, the following output shows the result of invoking the `mysql` client with no options, having executed the preceding command:

```
shell> mysql  
Welcome to the MySQL monitor. Commands end with ; or \g.  
...
```

## Server System Variables

- The MySQL server maintains many server system variables that indicate how it is configured.
- Each system variable has a default value.
- You can set variables at server startup by doing either of the following:
  - Use options on the command line.
  - Use an option file.
- You can refer to system variable values in expressions.
- You can view the values that a server uses.

- Use this command (at a shell prompt) to see the values that a server uses based on its compiled-in defaults and any option files that it reads:

```
mysqld --verbose --help
```

- Use this command (at a shell prompt) to see the values that a server uses based on its compiled-in defaults, ignoring the settings in any option files:

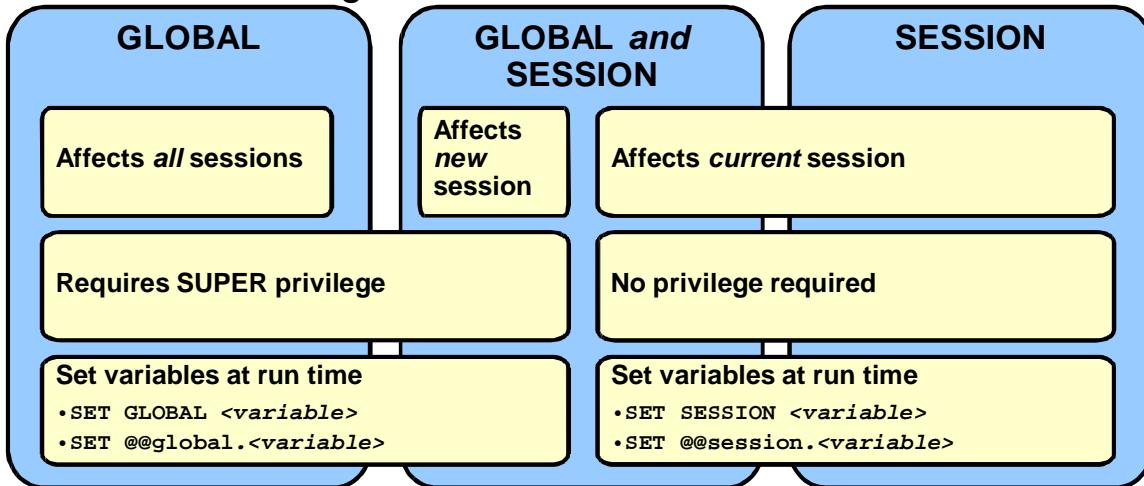
```
mysqld --no-defaults --verbose --help
```

- Within the mysql client, use this command to see only the variable values, without the additional startup options:

```
SHOW GLOBAL VARIABLES;
```

# Dynamic System Variables

- You can change most variables dynamically while the server is running.
  - In this way, you can modify the operation of the server without having to stop or restart it.
- Variable categories:



MySQL maintains two scopes that contain system variables. GLOBAL variables affect the overall operation of the server. SESSION variables affect its operation for individual client connections. Variables exist in one or the other scope, or in both.

Examples of variables and their scope include:

- **Global only:** key\_buffer\_size, query\_cache\_size
- **Both global and session:** sort\_buffer\_size, max\_join\_size
- **Session only:** timestamp, error\_count

When you change variable values, the following points apply:

- Setting a session variable requires no special privilege, but a client can change only its own session variables, not those of any other client.
- LOCAL and @@local are synonyms for SESSION and @@session.
- If you do not specify GLOBAL or SESSION, SET changes the session variable if it exists and produces an error if it does not.

# Displaying Dynamic System Variables

- List all available variables and their values:

```
SHOW [GLOBAL|SESSION] VARIABLES;
```

- List specific variable values:

```
mysql> SHOW VARIABLES LIKE 'bulk%';
+-----+-----+
| variable_name | Value |
+-----+-----+
| bulk_insert_buffer_size | 8388608 |
+-----+-----+
```

- Set a new value and then list:

```
mysql> SET bulk_insert_buffer_size=4000000;
mysql> SHOW VARIABLES LIKE 'bulk%';
+-----+-----+
| variable_name | Value |
+-----+-----+
| bulk_insert_buffer_size | 4000000 |
+-----+-----+
```

The specific SHOW VARIABLE session example in the slide uses 'bulk%' to find the variable for setting the insert buffer size, but any variable can be entered by using a whole variable name, or by entering a partial variable name with the percent sign (%) wildcard.

Specific variable types must be set as follows:

- Use a string value for variables that have a string type such as CHAR or VARCHAR.
- Use a numeric value for variables that have a numeric type such as INT or DECIMAL.
- Set variables that have a Boolean (BOOL or BOOLEAN) type to 0, 1, ON, or OFF. (If they are set on the command line or in an option file, use the numeric values.)
- Set variables of an enumerated type to one of the available values for the variable, but you can also set them to the number that corresponds to the desired enumeration value. For enumerated server variables, the first enumeration value corresponds to 0. This differs from ENUM columns, for which the first enumeration value corresponds to 1.

For more information about dynamic server variables, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/dynamic-system-variables.html>.

# Server Status Variables

- A structured variable differs from a regular system variable in two ways:
  - It is a structure with components that specify closely related server parameters.
  - There can be several instances of a given type of structured variable. Each one has a different name and refers to a different resource maintained by the server.
- MySQL supports a structured variable to govern the operation of key caches.

MySQL supports one structured variable type, which specifies parameters governing the operation of key caches. A key cache structured variable has these components:

```
key_buffer_size  
key_cache_block_size  
key_cache_division_limit  
key_cache_age_threshold
```

To refer to a component of a structured variable instance, you can use a compound name:

```
instance_name.component_name format
```

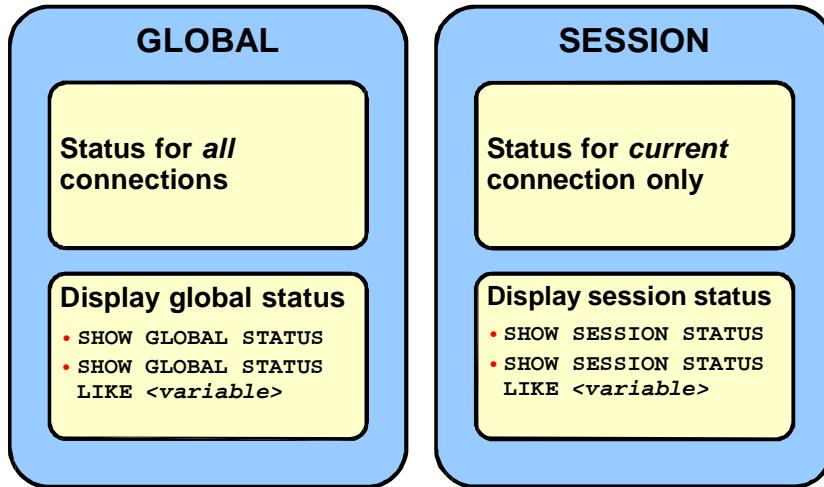
Examples: hot\_cache.key\_buffer\_size

```
hot_cache.key_cache_block_size
```

```
cold_cache.key_cache_block_size
```

# Server Status Variables

- Use the **SHOW STATUS** statement to assess the health of a system.
- Choose from two categories (similar to dynamic variables):



- LOCAL is a synonym for SESSION.
- If no modifier is present, the default is SESSION.

## SHOW STATUS Example:

```
mysql> SHOW GLOBAL STATUS;
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| Aborted_clients    | 0     |
| Aborted_connects   | 0     |
| Binlog_cache_disk_use | 0     |
| Bytes_received     | 169   |
| Bytes_sent          | 331   |
| Com_admin_commands  | 0     |
...

```

Some status variables have only a global value. For these, you get the same value for both **GLOBAL** and **SESSION**. For more information about server status variables, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/server-status-variables.html>.

## SQL Modes

Configure many server operation characteristics by setting the SQL mode:

- Specify how strict or forgiving MySQL is about accepting input data.
- Set compatibility with other database systems.
- Control query processing.
  - Enable or disable behavior relating to SQL conformance.
- Override SQL “empty” default mode.
  - Empty mode does not enable restrictions or conformance behaviors.

The default SQL mode is NO\_ENGINE\_SUBSTITUTION.

- The default configuration file adds STRICT\_TRANS\_TABLES.

The SQL mode consists of optional values that control some aspect of query processing. When you set the SQL mode appropriately, a client can have some control over the following:

- **Input data:** SQL modes can be used tell the server how strict or forgiving to be about accepting input data.
- **Standard SQL conformance:** SQL modes can be used to enable or disable behaviors relating to standard SQL conformance.
- **Compatibility:** SQL modes can be used to provide better compatibility with other database systems.

# Setting the SQL Mode

- Set at startup from the command line:

```
shell> mysqld --sql-mode=<mode_value>
```

- Set within an option file:

```
[mysqld]
sql-mode=IGNORE_SPACE
```

- SET statement

- Within mysql, after startup:

```
SET [SESSION|GLOBAL] sql_mode=<mode_value>
```

- Clear the current SQL mode:

```
SET sql_mode=''
```

Individual clients can configure the SQL mode for their own requirements, but you can also set the default SQL mode at server startup with the `--sql-mode` option. You may want to do this to run the server with a mode that is more cautious about accepting invalid data or creating MySQL user accounts.

If no modifier is present, `SET` changes the session SQL mode. You can invoke the `SET` statement with an empty string or with one or more mode names separated by commas. If the value is empty or contains more than one mode name, you must quote the values. If the value contains a single mode name, quoting is optional. SQL mode values are not case-sensitive.

Here are some `SET` examples:

- Set the SQL mode by using a single mode value:

```
SET sql_mode = ANSI_QUOTES;
SET sql_mode = 'TRADITIONAL';
```

- Set the SQL mode by using multiple mode names:

```
sql_mode = 'IGNORE_SPACE,ANSI_QUOTES,NO_ENGINE_SUBSTITUTION';
```

Check the current `sql_mode` setting by using this `SELECT` statement:

```
SELECT @@sql_mode;
```

# Common SQL Modes

|   |  |
|---|--|
| <b>STRICT_TRANS_TABLES, STRICT_ALL_TABLES</b> | <ul style="list-style-type: none"><li>Recommended for most installations</li><li>Sets “strict mode,” restricting acceptable database input</li><li>Bad values are treated as erroneous.</li></ul>      |
| <b>TRADITIONAL</b>                            | <ul style="list-style-type: none"><li>Sets “strict mode,” restricting acceptable input data values like other database servers</li><li>Restricts user account creation</li></ul>                       |
| <b>ANSI_QUOTES</b>                            | <ul style="list-style-type: none"><li>Sets double quote ("") as an identifier instead of a string-quoting character</li></ul>  |
| <b>IGNORE_SPACE</b>                           | <ul style="list-style-type: none"><li>Tells server to ignore spaces after function names</li><li>Allows space between name and the parenthesis</li><li>Sets function names as reserved words</li></ul> |
| <b>ERROR_FOR_DIVISION_BY_ZERO</b>             | <ul style="list-style-type: none"><li>Causes division-by-zero (when inserting table data) to produce a warning or an error</li></ul>   |
| <b>ANSI</b>                                   | <ul style="list-style-type: none"><li>Composite mode; allows standard SQL behavior</li><li>More “ANSI-like”</li></ul>  |
| <b>NO_ENGINE_SUBSTITUTION</b>                 | <ul style="list-style-type: none"><li>Disables automatic substitution of the default storage engine when the chosen engine for CREATE TABLE or ALTER TABLE is unavailable</li></ul>                    |

- **STRICT\_TRANS\_TABLES, STRICT\_ALL\_TABLES:** Without these modes, MySQL is forgiving with values that are missing, out of range, or malformed. Enabling STRICT\_TRANS\_TABLES sets “strict mode” for transactional tables; it is also enabled in the default my.cnf file. Enabling STRICT\_ALL\_TABLES sets strict mode for all tables.
- **TRADITIONAL:** Enable this SQL mode to enforce restrictions on input data values that are similar to those of other database servers. With this mode, using the GRANT statement to create users requires that you specify a password.
- **IGNORE\_SPACE:** By default, you must invoke functions with no space between the function name and the following parenthesis. Enabling this mode allows such spaces, and causes function names to be reserved words.
- **ERROR\_FOR\_DIVISION\_BY\_ZERO:** By default, division by zero produces a result of NULL. A division by zero when inserting data with this mode enabled causes a warning, or an error in strict mode.
- **ANSI:** Use this composite mode to cause the MySQL server to be more “ANSI-like.” That is, it enables behaviors that are more like standard SQL, such as ANSI\_QUOTES and PIPES\_AS\_CONCAT.
- **NO\_ENGINE\_SUBSTITUTION:** When you specify an unavailable storage engine while creating or altering a table, MySQL substitutes the default storage engine unless this mode is enabled. This is the default SQL mode.

# Log Files

- MySQL has different log files to record server activity:
  - Error
  - General Query
  - Slow Query
  - Binary
  - Enterprise Audit
- Log files:
  - Can use large amounts of disk space
  - Can be stored in files
  - Can be stored in tables
    - General and slow query logs only
  - Are written in text format
    - Except binary log

Record information about the SQL statements processed by the server:

- **Error log:** Diagnostic messages regarding startups, shutdowns, and abnormal conditions
- **General query log:** All statements that the server receives from clients
- **Slow query log:** Queries that take a long time to execute
- **Binary log:** Statements that modify data
- **Audit log:** Policy-based audit for Enterprise edition

Use these logs to assess the operational state of the server, for data recovery after a crash, for replication purposes, to help determine which queries are running slowly, and for security and regulatory compliance. None of these logs is enabled by default (except the error log in Windows). It is important to understand that log files, particularly the general query log, can grow to be quite large.

For more information about log files, see the *MySQL Reference Manual*:

<http://dev.mysql.com/doc/mysql/en/server-logs.html> and

<http://dev.mysql.com/doc/mysql/en/mysql-enterprise-audit.html>

## Log File Usage Matrix

| Log File   | Options                                | File Name            | Programs      |
|------------|--|----------------------|---------------|
|            |  | Table Name           |               |
| Error      | --log-error                            | host_name.err        | N/A           |
| General    | --general_log                          | host_name.log        | N/A           |
|            |  | general_log          |               |
| Slow Query | --slow_query_log<br>--long_query_time  | host_name-slow.log   | mysqldumpslow |
|            |  | slow_log             |               |
| Binary     | --log-bin<br>--expire-logs-days        | host_name-bin.000001 | mysqlbinlog   |
| Audit      | --audit_log<br>--audit_log_file<br>... | audit.log            | N/A           |

The server creates all the log files in the data directory and sets the file name to the current hostname if you do not set another path name. You can enable these log files by starting the server with the corresponding options (on the command line or in an option file without the preceding '--').

- **Error log:** Set `--log-error=<file_name>` to log errors to the given file. The `mysqld_safe` script creates the error log and starts the server with its output redirected to the error log.
- **General query log:** Set `--general_log_file=<file_name>` to log queries. The global `general_log` and `general_log_file` server variables provide runtime control over the general query log. Set `general_log` to 0 (or OFF) to disable the log, or to 1 (or ON) to enable it.
- **Slow query log:** Set `--slow_query_log_file=<file_name>` to provide runtime control over the slow query log. Set `slow_query_log` to 0 to disable the log, or to 1 to enable it. If a log file is already open, it is closed and the new file is opened.

If the general or slow query logs are enabled, they can output log entries to a file, a table in the `mysql` database, or both, by setting the `--log-output` option. See the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/log-destinations.html>

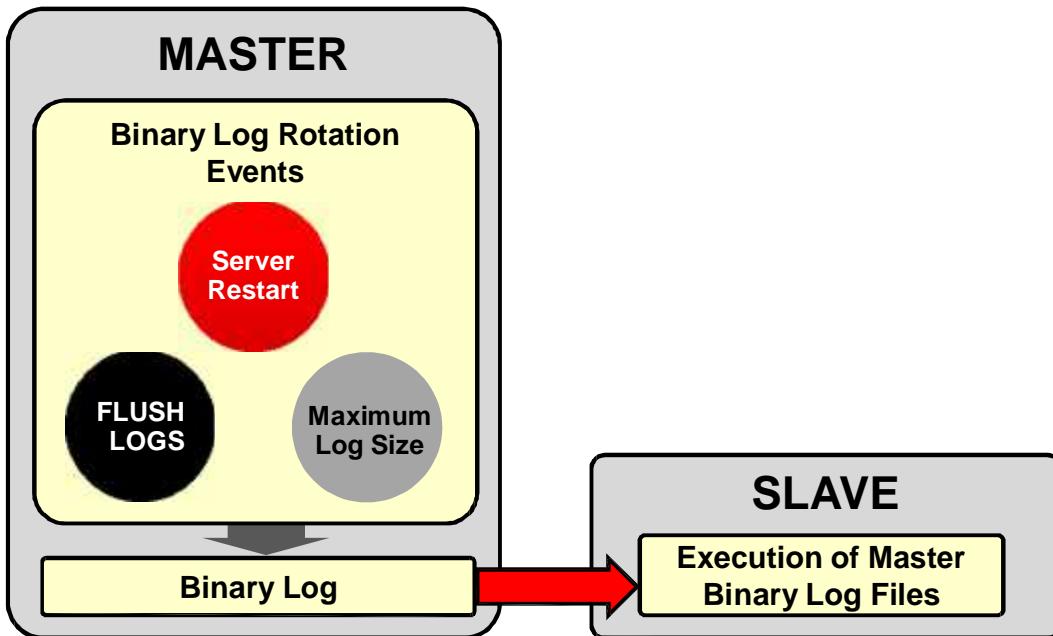
- **Binary log:** Set `--log-bin` to enable binary logging. The server uses the option value as a base name, adding an increasing sequential numeric suffix to the base name as it creates new log files. These log files are stored in binary format rather than text format.

For detailed information on the options available for all of the above log types, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/server-logs.html>

- **Audit log:** The audit log is provided as an Enterprise Edition plugin that, when loaded, you can use to log events into the file specified by the `audit_log_file` option. Auditing constantly writes to the audit log until you remove the plugin, or you turn off the auditing with the `audit_log_policy=NONE` option setting. You can prevent the removal of the plugin by using `audit_log=FORCE_PLUS_PERMANENT` as an option when the server is started. For detailed information about all available audit plugin options, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/audit-log-plugin-options-variables.html>

# Binary Logging

Log shipping system:



The binary log contains “events” that describe database changes such as table creation or changes to table data. It also contains events for statements that could have potentially made changes (for example, a `DELETE` that matched no rows). It also contains information about how long each update statement took. The binary log has two important purposes: replication and data recovery.

MySQL uses a log-shipping replication solution. With the log-shipping system, you can store all data changes that occur on the *master* in a *binary log* and then retrieve them with the slave, and execute from these received log files. You can download log files and execute the contents in real time; that is, as soon as a log file event is generated, it is sent to the connected slaves for execution. Due to network propagation delays, it can take from a few seconds to a few minutes (worst case) for the slave to receive the updates. In ideal situations, the delay is well under one second.

The binary log rotates when one of the following events occurs:

- The MySQL server is restarted.
- The maximum allowed size is reached (`max_binlog_size`).
- A `FLUSH LOGS` SQL command is issued.

The binary log is independent of the storage engine. MySQL replication works regardless of the storage engine that is being used (that is, InnoDB or MyISAM).

**Note:** The data recovery usage of binary logs is covered in detail later in the course.

# Binary Logging Formats

Choose from two different types of logging:

|                             | Statement-Based  | Row-Based  |
|-----------------------------|--|--|
| Size of log files           | Small  | Large  |
| Replication limitations     | Not all statements can be replicated.  | All statements can be replicated.  |
| Master/Slave MySQL versions | Slave can be a newer version with a different row structure.                       | Slave must be an identical version and row structure.  |
| Locking                     | <code>INSERT</code> and <code>SELECT</code> require a greater number of row locks. | <code>INSERT</code> , <code>UPDATE</code> , and <code>DELETE</code> require fewer locks on slaves. |
| Point-in-time recovery      | Yes  | Yes (more difficult due to binary format of log events)  |

Statement-based binary logging:

- Contains the actual SQL statements
- Includes both DDL (`CREATE`, `DROP`, and so on) and DML (`UPDATE`, `DELETE`, and so on) statements
- The relatively small files save disk space and network bandwidth.
- Not all replicated statements replay correctly on a remote machine.
- Requires that replicated tables and columns be identical (or compatible with several restrictions) on both master and slave

Row-based binary logging:

- Indicates how individual table rows are affected
- Replays all statements correctly, even for changes caused by features that do not replicate correctly when using statement-based logging

Set the format as follows:

```
SET [GLOBAL|SESSION] BINLOG_FORMAT=[row|statement|mixed|default];
```

**Note:** Use the `mixed` option to cause MySQL to pick the most appropriate format for individual events. It generally uses the statement-based binary log, but it reverts to the row-based replication when required.

## List Binary Log Files

- Use the **SHOW BINARY LOGS** statement to:
  - List current log files and file size

```
mysql> SHOW BINARY LOGS;
+-----+-----+
| Log_name      | File_size |
+-----+-----+
| binlog.000015 |    724935 |
| binlog.000016 |    733481 |
+-----+-----+
```

- Use the **SHOW MASTER STATUS** statement to:
  - Show the master status for the next event

```
mysql> SHOW MASTER STATUS;
+-----+-----+-----+-----+
| File      | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| binlog.000016 |    733481 | world_innodb |      manual,mysql |
+-----+-----+-----+-----+
```

Statements are stored in the binary logs in the form of “events” that describe the database modifications. To identify an event, you must have the binary log file name and the byte offset (or position). For example, the log file `binlog.000016`, position `733481` would identify the last event in the slide.

The `SHOW MASTER STATUS` command requires either the `SUPER` or the `REPLICATION CLIENT` privilege.

## View Binary Log Contents

- Cannot be viewed with normal text viewers:
  - Logs are stored in compact binary format.
- Use the **mysqlbinlog** utility to:
  - Convert binary logs to SQL text format
- View the data in standard out:

```
shell> mysqlbinlog host-bin.000001 host-bin.000002
```

- Redirect output to `more`:

```
shell> mysqlbinlog host-bin.000001 | more
```

The `mysqlbinlog` program is executed from the command line, specifying as parameters the logs to be viewed. There are several options for this program, providing features such as the ability to extract events based on time or file offset. You can view all options with the `--help` flag.

In a `mysqlbinlog` output, events are preceded by header comments that provide information:

```
# at 141
#100309  9:28:36 server id 123  end_log_pos 245
Query thread_id=3350  exec_time=11  error_code=0
```

If you create a binary log by using statement-based logging, and if you then run `mysqlbinlog --database=test` to create a readable file, the server filters out statements that are not associated with the `test` database:

```
INSERT INTO test.t1 (i) VALUES(100);
-----
| USE test;                                |
| INSERT INTO test.t1 (i) VALUES(101);      |
| INSERT INTO t1 (i)          VALUES(102);   |
| INSERT INTO db2.t2 (j)  VALUES(201);      |
-----
USE db2;
INSERT INTO db2.t2 (j)  VALUES(202);
```

The first `INSERT` statement is not included because there is no default database. It outputs the three `INSERTs` following `USE test`, but not the `INSERT` following `USE db2`.

## Deleting Binary Logs

- By default, old log files are not deleted.
- Delete logs based on age:

```
SET GLOBAL expire_logs_days = 7;  
...or...  
PURGE BINARY LOGS BEFORE now() - INTERVAL 3 day;
```

- Delete logs based on file name:

```
PURGE BINARY LOGS TO 'mysql-bin.000010';
```

To automatically delete any binary logs older than a specific number of days during a binary log rotation, use the `expire_logs_days` setting.

You can also configure `expire_logs_days` in the option file:

```
[mysqld]  
expire_logs_days=7
```

# Configuring Enterprise Audit

- Implemented using the **audit\_log** server plugin
  - Available with Enterprise Edition subscriptions
- Policy-based logging:
  - Is set with the `audit_log_policy` option
  - Offers the choice of logging ALL, NONE, LOGINS, or QUERIES
  - Defaults to ALL
- Produces an audit record of server activity in a log file
  - Contents depend on policy, and can include:
    - A record of errors that occur on the system
    - When clients connect and disconnect
    - What actions they perform while connected
    - Which databases and tables they access

To install the `audit_log` plugin, use the `INSTALL PLUGIN` syntax, as in the following example:

```
INSTALL PLUGIN audit_log SONAME 'audit_log.so';
```

Alternatively, set the `plugin-load` option at server startup:

```
[mysqld]
plugin-load=audit_log.so
```

By default, loading the plugin enables logging. Setting the option `audit_log` to OFF disables logging. To prevent the plugin from being removed at run time, set the following option:

```
audit_log=FORCE_PLUS_PERMANENT
```

The log file is named `audit.log`, and by default is in the server data directory. To change the name or location of the file, set the `audit_log_file` system variable at server startup.

To balance compliance with performance, use the `audit_log_strategy` option to choose between SYNCHRONOUS, ASYNCHRONOUS, SEMISYNCHRONOUS, and PERFORMANCE.

If you set the `audit_log_rotate_on_size` option to some number greater than 0, the log file is rotated when its size exceeds that number of 4 KB blocks.

For more information about configuring Enterprise Audit, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/audit-log-plugin-options-variables.html>

# Audit Log File

The audit log file is written as XML, using UTF-8.

- The root element is <AUDIT>.
- The closing </AUDIT> tag of the root element is written when the plugin terminates.
  - The tag is not present in the file while the audit log plugin is active.
- Each audit entry is an <AUDIT\_RECORD /> element:

```
<AUDIT_RECORD TIMESTAMP="2012-10-12T09:35:15"
  NAME="Connect" CONNECTION_ID="4" STATUS="0"
  USER="root" PRIV_USER="root" OS_LOGIN="" PROXY_USER=""
  HOST="localhost" IP="127.0.0.1" DB=""/>
<AUDIT_RECORD TIMESTAMP="2012-10-12T09:38:33"
  NAME="Query" CONNECTION_ID="4" STATUS="0"
  SQLTEXT="INSERT INTO tbl VALUES(1, 2)"/>
```

The `TIMESTAMP` of each audit record is in UTC.

The `NAME` attribute represents the type of event. For example, “Connect” indicates a login event, “Quit” indicates a client disconnect, and “Shutdown” indicates a server shutdown. “Audit” and “NoAudit” indicate the points at which auditing starts and stops.

The `STATUS` attribute provides the command status. This is the same as the `Code` value displayed by the MySQL command `SHOW ERRORS`.

Some attributes appear only for specific event types. For example, a “Connect” event includes attributes such as `HOST`, `DB`, `IP`, and `USER`, and a “Query” event includes the `SQLTEXT` attribute.

For more information about the Enterprise Audit log file, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/audit-log-file.html>

# Log File Rotation

- Log files take up space over time. Periodically back up and remove old log files and recommence logging to new log files.
  - Use caution if you are using binary logs for replication.
- After backup, flush the logs. Flushing the logs:
  - Creates new binary log files
  - Closes and re-opens the general and slow query log files
    - To create new logs, you must rename the current log files before flushing.
- Schedule log file rotation at regular intervals:
  - Create your own script or use the provided **mysql-log-rotate** script (RHEL only)

If you are using a Red Hat Enterprise-based operating system such as Oracle Linux, use the `mysql-log-rotate` script in `/usr/share/mysql/`, provided as part of the RPM installation.

After backing up all the log files, flush the logs to force the MySQL server to start writing to new log files. Execute the `FLUSH LOGS` SQL statement, or run a suitable client command. Flushing the logs causes binary logging to recommence with the next file in the sequence. With the general and slow query logs however, flushing merely closes the log files, reopens them and then recommences logging under the same file name. To start new logs, rename the existing log files before flushing. Example:

```
# cd mysql-data-directory
# mv mysql.log mysql.old
# mv mysql-slow.log mysql-slow.old
# mysqladmin flush-logs
```

For more information about managing log files, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/log-file-maintenance.html>.

# Command-Line Client Programs

| Prompt                   | Description                                   |
|--------------------------|---|
| <code>mysql</code>       | Sends SQL statements to the server            |
| <code>mysqladmin</code>  | Assists in managing the server                |
| <code>mysqlimport</code> | Loads data files into tables                  |
| <code>mysqldump</code>   | Backs up the contents of databases and tables |

The `mysqld` server program must be running for clients to gain access to databases.

`mysql` is a general-purpose command-line client for sending SQL statements to the server, including those of an administrative nature.

`mysqladmin` is an administrative command-line client that assists in managing the server.

`mysqlimport` provides a command-line interface to the `LOAD DATA INFILE` statement. It is used to load data files into tables without having to issue `LOAD DATA INFILE` statements manually.

`mysqldump` is a command-line client for dumping the contents of databases and tables. It is useful for making backups or for copying databases to other machines.

The list in the slide includes only a few of the command-line programs in a MySQL installation. This course provides more details for the DBA-related programs in this and later lessons.

For more information about command-line client programs, see the *MySQL Reference Manual*:

<http://dev.mysql.com/doc/mysql/en/programs-client.html>

<http://dev.mysql.com/doc/mysql/en/programs-admin-utils.html>

# Invoking Command-Line Clients

- Invoke from a command line:
  - Linux or UNIX shell prompt (terminal window)
  - Windows console prompt

```
shell> <client> [options]
```

- Determine supported options:

```
shell> <client> --help (or -?)
```

- Determine the version of the client program:

```
shell> <client> --version (or -v)
```

- Two general forms of option syntax:
  - Long option (`--<option>`)
  - Short option (`-<option>`)

You control the behavior of client programs by specifying options following the program name.

Find out how to use the `mysql` client:

```
shell> mysql --help
...
-?, --help           Display this help and exit.
-I, --help           Synonym for -?

...
--character-sets-dir=name   Directory for character set files.
--column-type-info    Display column type information.
-c, --comments       Preserve comments. Send comments to the server. The default
                     is --skip-comments (discard comments), enable with --comments
-C, --compress        Use compression in server/client protocol.
...
```

Determine the version of the MySQL distribution that you are running:

```
shell> mysql -v
mysql Ver 14.14 Distrib 5.6.10, for Linux (x86_64) using EditLine wrapper
```

The client programs that you run do not have to be the same version as the server.

# Connection Parameter Options

- Connect with host indicated:
  - Locally, to server running on same host
  - Remotely, to server running on different host
- Common client-specific connection options:
  - `-h <host_name>` or `--host=<host_name>`
  - `-C` or `--compress`
  - `--protocol=<protocol_name>`
  - `-P <port_number>` or `--port=<port_number>`
  - `-S <socket_name>` or `--socket=<socket_name>`
  - `--shared-memory-base-name=<shared_memory_name>`

These are a few of the most common connection parameter options:

- `-h`: Followed by the name or IP address of a given host, to connect to the server (the default is `localhost`)
- `-c`: Compresses all information sent between the client and the server if both support compression
- `--protocol`: Followed by the connection protocol to use for connecting to the server: {TCP|SOCKET|PIPE|MEMORY}
- `-P`: Followed by a port number to use instead of the default (3306)
- `-S`: Sets UNIX socket file, or the name of the named pipe to use on Windows
- `--shared-memory-base-name`: (Windows only) The shared-memory name to use for connections that are made using shared memory to a local server. This option applies only if the server supports shared-memory connections.

Examples of how you can use some of these options:

```
mysql -h 192.168.1.101 -P 3351 -u root -p
mysql --host=localhost --compress
mysql --host=localhost -S /var/lib/mysql/mysql.sock
mysql --protocol=TCP
```

## **Quiz**

All of the MySQL command-line clients display a list of available options when the client command is executed with the \_\_\_\_\_ option.

- a. -v
- b. --help
- c. --List
- d. --options

# Invoking mysql Client

- Provide credentials on the command line:

```
shell> mysql -u<name> -p<password>
```

- Provide credentials in a login path:

```
shell> mysql --login-path=<login-path>
```

- Execute a statement:

```
shell> mysql --login-path=<login-path> -e "<statement>"
```

- Execute using a specific option file:

```
shell> mysql --defaults-file=<opt_file_name> ...
```

- Execute using a text file containing SQL statements:

```
shell> mysql ... < <file_name.sql>
```

- **-u<name>** (or **--username=<name>**): With or without a space after the option.
- **-p<password>** (or **--password=<password>**): Without a space after the option. When you use the option without a value, you are prompted for the password. You can also put it in an option file instead of on the command line, or provide credentials in a login path.
- **--login-path=<login-path>**: Use the credentials for this login path as created with **mysql\_config\_editor**.
- **-e "<statement>"** (or **--execute=<statement>**): Invokes the **mysql** client and then executes the SQL statement. For example, to see the current server version:

```
shell> mysql --login-path=admin -e "SELECT VERSION( )"
+-----+
| VERSION()           |
+-----+
| 5.6.10-enterprise-commercial-advanced-log |
+-----+
```

- Redirect output to a file by adding **> <file\_name>**.
- Run a “script” or “batch” file by adding **< <file\_name>**.
  - File must be in plain text format with a statement terminator for each statement.
  - File must be located on the host that is running the **mysql** client.
- **<** and **-e** are mutually exclusive. They cannot be used together.

## mysql Client: Safe Updates

- Avoid statements that make extreme modifications.
  - Add the **--safe-updates** option when invoking the client:

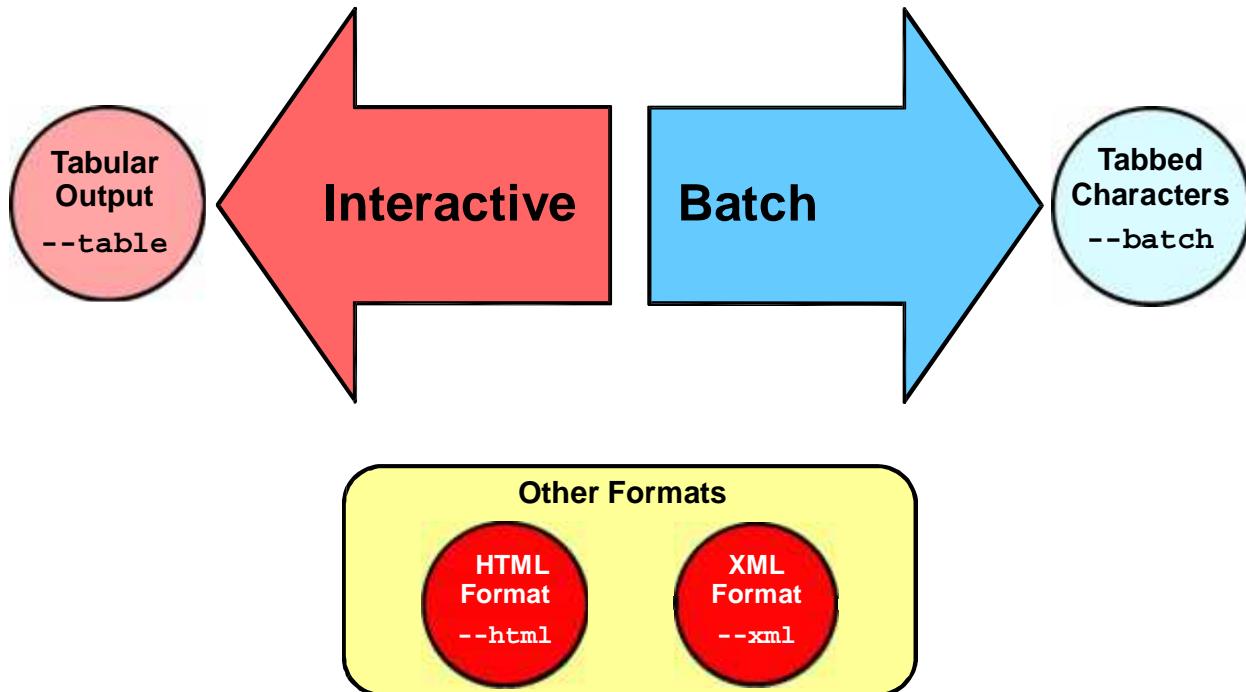
```
shell> mysql ... --safe-updates
```

- Protect users from issuing potentially dangerous statements:
  - Useful for MySQL beginners
  - UPDATE and DELETE are allowed only with WHERE or LIMIT.
    - WHERE must specify which records to modify, using a key value.
  - SELECT output is restricted.

You can inadvertently issue statements that modify many rows in a table or statements that return extremely large result sets. The **--safe-updates** option helps prevent these problems. Set safe updates mode to impose the following SQL statement restrictions:

- UPDATE and DELETE are allowed only if they include a WHERE clause that specifically identifies which records to update or delete by means of a key value, or if they include a LIMIT clause.
- Output from single-table SELECT statements is restricted to no more than 1,000 rows unless the statement includes a LIMIT clause.
- Multiple-table SELECT statements are allowed only if MySQL examines no more than 1,000,000 rows to process the query.

## mysql Client: Output Formats



By default, mysql produces output whether it is used in interactive or batch mode:

- **Interactive:** When you invoke mysql interactively, it displays query output in a tabular format that uses bars and dashes to display values lined up in boxed columns.
  - **--table (or -t):** Produces tabular output format even when running in batch mode. This is the default format for interactive mode.
- **Batch:** When you invoke mysql with a file as its input source on the command line, it runs in batch mode with query output displayed using tab characters between data values.
  - **--batch (or -B):** Produces batch mode (tab-delimited) output, even when running interactively, and does not use history file. This is the default format for batch mode.
  - In batch mode, use the **--raw** or **-r** options to suppress conversion of characters such as newline and carriage return to escape sequences such as **\n** or **\r**. In raw mode, the characters are printed literally.

Use these options to select an output format that is different from either of the default formats:

- **--html (or -H):** Produces output in HTML format
- **--xml (or -X):** Produces output in XML format

# **mysql Client: MySQL Client Commands**

- List all MySQL client-level commands:

```
mysql> HELP
```

- Display session status information:

```
mysql> STATUS
```

- Log session queries and their output:

```
mysql> tee my_tee_file.txt
```

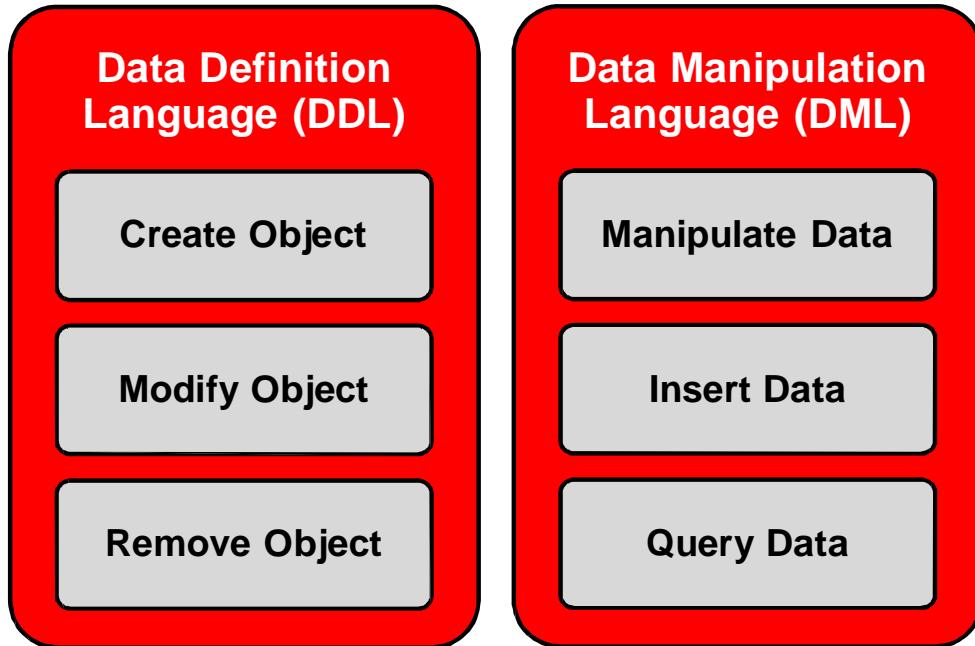
Using interactive mode for typical day-to-day usage, you can submit commands that display information, as in this example:

```
mysql> HELP
...
?          (\?) Synonym for `help'.
clear      (\c) Clear the current input statement.
connect    (\r) Reconnect to the server. Optional arguments are db and host.
delimiter  (\d) Set statement delimiter.
ego        (\G) Send command to mysql server, display result vertically.
exit       (\q) Exit mysql. Same as quit.
...
```

You can display specific information about the current client session:

```
mysql> STATUS
mysql Ver 14.14 5.6.8, for Linux (x86_64) using EditLine wrapper
Connection id:          8
Current database:       world_innodb
Current user:           root@localhost
SSL:                  Not in use
Using delimiter:         ;
Server version: 5.6.8-enterprise-commercial-advanced-log MySQL Enterprise
Server - Advanced Edition (Commercial)
...
```

# `mysql` Client: SQL Statements



If you issue an SQL statement on a system running `mysql`, the client sends this statement to the MySQL server and the server executes the statement and returns the results.

## **Data Definition Language (DDL)**

Some common SQL statements:

- **`CREATE DATABASE/TABLE`**: Creates a database or table with the given name
- **`ALTER DATABASE/TABLE`**: Can change the overall characteristics of a database or table
- **`DROP DATABASE/TABLE`**: Drops all tables in the database and deletes the database, or drops a specific table

## **Data Manipulation Language (DML)**

Some common SQL statements:

- **`SELECT`**: Retrieves rows selected from one or more tables
- **`INSERT`**: Inserts new rows into an existing table
- **`DELETE`**: Deletes rows from an existing table
- **`UPDATE`**: Updates columns of existing rows in the named table with new values
- **`JOIN`**: Combines tables as part of `SELECT`, multiple-table `DELETE` and `UPDATE` statements

## **mysql Client: Help on SQL Statements**

- View full SQL category list:

```
mysql> HELP CONTENTS
...
      Account Management
      Administration
      Compound Statements
      Data Definition
      Data Manipulation
      Data Types
...
```

- Help on a specific SQL category or statement:

```
mysql> HELP Data Manipulation
mysql> HELP JOIN
```

- Help on status-related SQL statements:

```
mysql> HELP STATUS
```

You can access server-side help within the mysql client. Server-side help performs lookups in the *MySQL Reference Manual* for a particular topic, directly from the mysql> prompt. Use HELP followed by a keyword to access information. To display the top-most entries of the help system, use the CONTENTS keyword.

You do not have to step through the items listed in the contents list to get help on a specific subject. Just give a topic as the keyword to get some hints. For example, HELP STATUS results in a list of status-related SQL statements:

```
...
SHOW
SHOW ENGINE
SHOW MASTER STATUS
...
```

For more information about the HELP statement, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/help.html>.

# mysql Client: SQL Statement Terminators

- Common SQL terminators
  - ; or \g
  - \G (vertical display of output)

```
mysql> SELECT VERSION();
+-----+
| VERSION() |
+-----+
| 5.6.10-enterprise-commercial-advanced-log |
+-----+
1 row in set (0.00 sec)

mysql> SELECT VERSION()\G
***** 1. row *****
VERSION(): 5.6.10-enterprise-commercial-advanced-log
1 row in set (0.00 sec)
```

- Abort statement
  - Use the \c terminator

```
mysql> SELECT VERSION()\c
mysql>
```

SQL statements require a terminator:

- ; and \g: Common terminators that are equivalent and can be used interchangeably
- \G: Terminates queries and displays query results in a vertical style that shows each output row with each column value on a separate line. This terminator is especially useful if a query produces very wide output lines because the vertical format can make the result much easier to read.
- \c: If you decide to abandon a statement that you are composing, you can cancel the statement and return to a new mysql> prompt.

## mysql Client: Special Statement Terminators

- Use multi-line statements:
  - Terminator is required at end.
  - Prompt changes from `mysql>` to `->`.

```
mysql> SELECT VERSION(),  
-> DATABASE();
```

- End the session and exit the mysql client:
  - Use the `\q` terminator or `QUIT` or `EXIT`.

```
mysql> \q  
Bye
```

With mysql, you can enter a single query over multiple input lines. This makes it easier to issue a long query, because you can enter it over the course of several lines. mysql waits until it sees the statement terminator before sending the query to be executed, as in this example:

```
mysql> SELECT Name, Population FROM City  
-> WHERE CountryCode = 'IND'  
-> AND Population > 3000000;
```

If a statement results in an error, mysql displays the error message returned by the server:

```
mysql> This is an invalid statement;  
ERROR 1064 (42000): You have an error in your SQL syntax ; check  
the manual that corresponds to your MySQL server version ...
```

Additional commands:

- `edit (\e)`: Edit command with \$EDITOR.
- `pager (\P)`: Set PAGER [to\_pager]. Print the query results via PAGER.
- `rehash (\#)`: Rebuild completion hash.

These additional commands work on UNIX and Linux operating systems but are not supported on Windows.

## mysql Client: Redefining the Prompt

- Redefine the prompt:

```
mysql> PROMPT term 1>
      term 1>
```

- Add information within the prompt:

```
mysql> PROMPT(`\u@\h` [\d]\>
PROMPT set to `(\u@\h) [\d]\>`
(root@localhost) [test]>
```

- Return to the original prompt:

```
(root@localhost) [test]>
mysql> PROMPT
mysql>
```

The mysql> prompt is the primary (or default) prompt. It signifies that the mysql client is ready for a new statement to be entered.

You can change the default prompt by placing current information in the prompt, such as the user (\u), host (\h), and database (\d), as shown in the example in the slide.

**Note:** The second example in the slide assumes that the database was previously set to test.

Everything following the first space after the PROMPT keyword becomes part of the prompt string, including other spaces. The string can contain special sequences. To return the prompt to the default, specify PROMPT or \R with no arguments.

## **mysql Client: Using Script Files**

- Process input files from within `mysql`:
  - If the files contain SQL statements, they are known as:
    - “Script file”
    - “Batch file”
- Use the `SOURCE` command:

```
mysql> SOURCE /usr/stage/world_innodb.sql
Query OK, 0 rows affected (0.00 sec)
...
...
```

When you run it interactively, `mysql` reads queries entered at the keyboard. `mysql` also accepts input from a file. The MySQL server executes the queries in the file and displays any output produced. An input file containing SQL statements to be executed is known as a “script file” or a “batch file.” A script file should be a plain-text file containing statements in the same format that you would use to enter the statements interactively. In particular, each statement must end with a terminator.

Quotation marks are not required around the name of the file following the `SOURCE` command.

## **mysqladmin Client**

- A command-line client for DBAs
- Many capabilities
  - “Ping” the server.
  - Shut down the server.
  - Create and drop databases.
  - Display server and version information.
  - Display or reset server status variables.
  - Set passwords.
  - Reload grant tables.
  - Flush log files and caches.
  - Start and stop replication.
  - Display client information.

You can check the server's configuration and current status, create and drop databases, and more by using the `mysqladmin` command-line client.

**Note:** `mysqladmin` is covered later in this course in regard to security and user management.

## Invoking the mysqladmin Client

- Execute from a command-line prompt:

```
shell> mysqladmin --login-path=<login-path> commands  
shell> mysqladmin -u<name> -p<password> commands
```

- Get full list of options:

```
shell> mysqladmin --help
```

- Examples:

```
shell> mysqladmin --login-path=admin processlist  
shell> mysqladmin -uroot -poracle status variables  
shell> mysqladmin --login-path=admin shutdown
```

mysqladmin accepts one or more commands following the program name. For example, the command options displayed in the slide have the following results:

- **status:** Displays a brief status message, followed by the list of server variables

```
shell> mysqladmin --login-path=admin status  
Uptime: 363501 Threads: 1 Questions: 5441 Slow queries: 0 Opens: 53  
Flush tables: 1 Open tables: 0 Queries per second avg: 0.14
```

- **variables:** Displays server system variables and their values

```
shell> mysqladmin -uroot -poracle variables  
Warning: Using a password on the command line interface can be insecure.  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| auto_increment_increment | 1 |  
| auto_increment_offset | 1 |  
| autocommit | ON |  
| automatic_sp_privileges | ON |  
| back_log | 50 |  
| basedir | /usr |  
... |
```

- **processlist**: Lists active server threads

```
shell> mysqladmin --login-path=admin processlist
+----+-----+-----+-----+-----+-----+
| Id | User | Host          | db | Command | Time | State | Info
+----+-----+-----+-----+-----+-----+
| 35 | root | localhost:1184 |   | Query   | 0    |       | show processlist |
+----+-----+-----+-----+-----+-----+
```

- **shutdown**: Stops the server

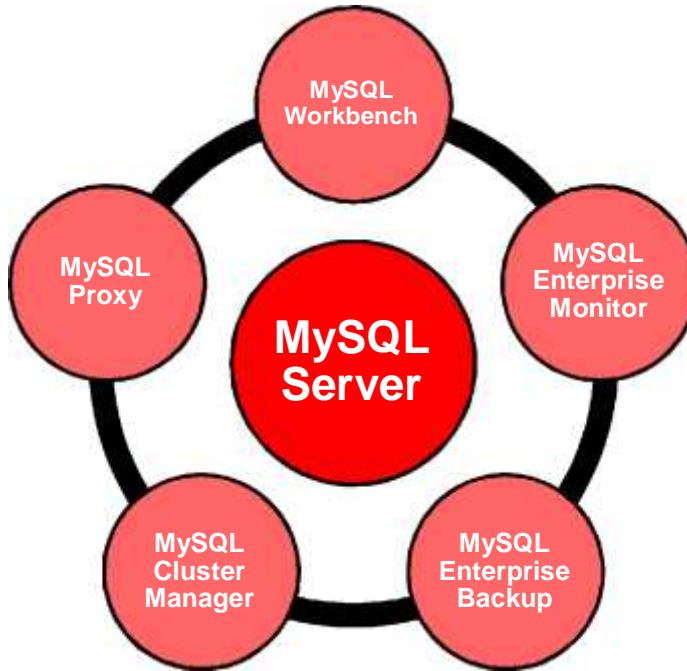
Some `mysqladmin` commands are available only to MySQL accounts that have the required privileges. For example, to shut down the server, you must connect to it using an administrative account such as `root` that has the `SHUTDOWN` privilege.

## **Quiz**

SQL statements can be issued within the mysqladmin client for creation and manipulation of data by using statements such as CREATE DATABASE, SELECT, and ALTER TABLE.

- a. True
- b. False

# MySQL Tools



Oracle Corporation provides tools for administering and otherwise working with the MySQL Server.

- **MySQL Workbench:** A next-generation visual database design application that can be used to efficiently design, manage, and document database schemata. It is available in both open-source and commercial editions.
- **MySQL Proxy:** A simple program that sits between your client and MySQL server(s) that can monitor, analyze, or transform their communication. MySQL Proxy's flexibility allows for a wide variety of uses, including load balancing, failover, query analysis, query filtering and modification, and many more. Currently it is not production-ready.

The following tools are available only in select MySQL Commercial Editions:

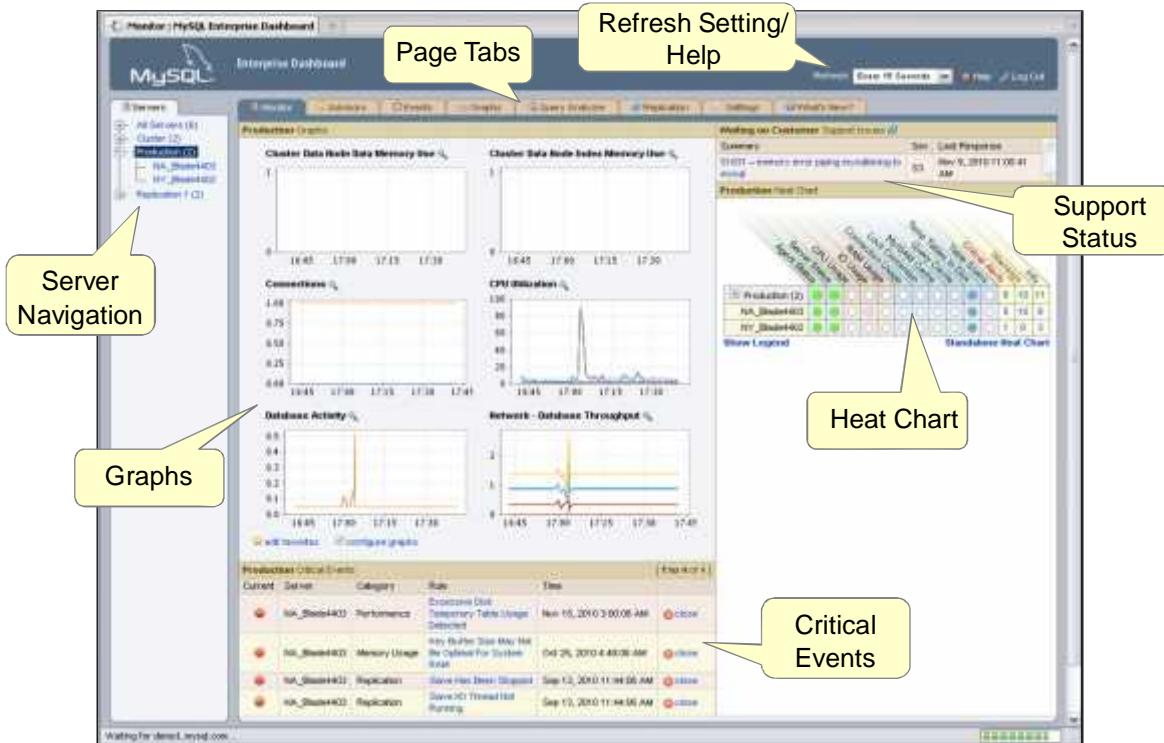
- **MySQL Enterprise Monitor:** A visual enterprise-monitoring system for MySQL that observes your MySQL servers, notifies you of potential issues and problems, and advises you how to fix the issues
- **MySQL Enterprise Backup:** Enables you to perform online “hot,” non-blocking backups, restores your data from a full backup, and supports creating compressed backup files
- **MySQL Cluster Manager:** Simplifies the creation and management of the MySQL Cluster Carrier Grade Edition database by automating common management tasks

# MySQL Enterprise Monitor

- Web-based monitoring and advising system
  - Virtual DBA assistant
  - Runs completely within a corporate firewall
- Principle features:
  - Enterprise Dashboard
    - “At-a-glance” monitoring
  - Server/group management
  - MySQL and custom advisors and rules
  - Advisor rule scheduler
  - Customizable thresholds and alerts
  - Events and alert history
  - Replication Monitor
  - Query Analyzer

Enterprise Monitor helps you manage more MySQL servers in a multinode (horizontally scaled) environment, tune your current MySQL servers, and find and fix problems with your MySQL database applications before they can become serious problems or costly outages. This web GUI application proactively monitors enterprise database environments and provides expert advice on how MySQL can tighten security, optimize performance, and reduce the down time of MySQL-powered systems. Enterprise Monitor monitors all kinds of configurations, from a single MySQL server all the way up to a huge farm of MySQL servers powering a busy website.

# MySQL Enterprise Monitor: Dashboard



The rich GUI Enterprise Dashboard contains tabbed “pages” that provide an immediate graphic view of administrative tasks, server and database status, and advisory information.

## **MySQL Enterprise Monitor: Access**

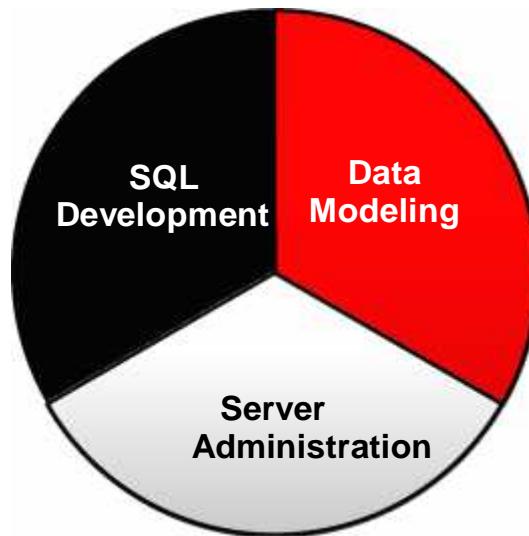
- Commercial product
- “Trial” version available
- Prerecorded demos
- <http://www.mysql.com/products/enterprise/monitor.html>

For more information about the Enterprise Monitor tool, refer to the MySQL website:  
<http://www.mysql.com/products/enterprise/monitor.html>.

This site also provides demos.

# MySQL Workbench

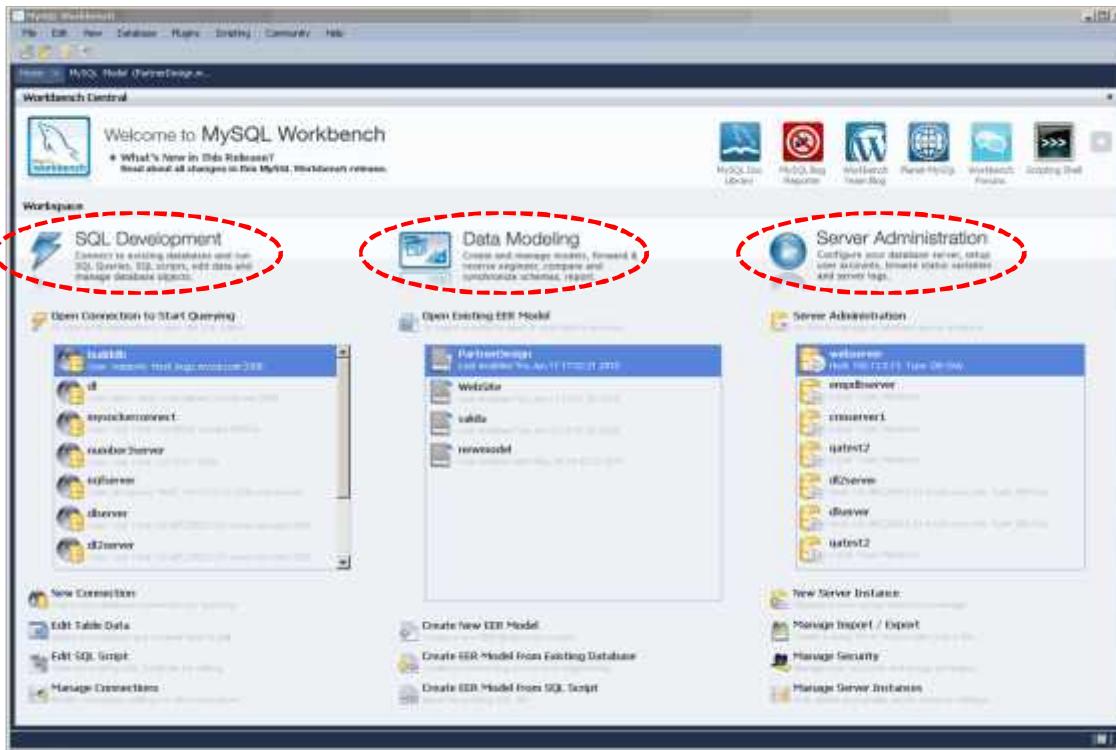
- GUI tool for managing the MySQL server
- Three separate functional modules:



MySQL Workbench provides DBAs and developers with a GUI-based, cross-platform, integrated-tools environment for three primary functions.

- **SQL Development** facilitates the following tasks:
  - Edit and execute SQL queries and scripts.
  - Create or alter database objects.
  - Edit table data.
- **Database Design and Modeling** facilitates the following tasks:
  - Perform enhanced entity relationship (EER) modeling.
  - Edit and execute SQL queries and scripts.
  - Design, generate, and manage databases.
- **Server Administration** (replaces MySQL Administrator) facilitates the following tasks:
  - Start and stop the server.
  - Edit database server configuration.
  - Manage users.
  - Import and export data.

# MySQL Workbench: GUI Window



When you invoke MySQL Workbench, the top-level GUI opens to show the current information for each module. Click the module title to go into that specific tool and complete tasks.

# MySQL Workbench: Access

- Two versions of Workbench
    - Community Edition (OSS)
    - Standard Edition (SE)
  - OS platforms
    - Windows
    - Linux
    - Mac OS X
  - Installation
    - Requirements in *MySQL Workbench Reference Manual*
    - Download webpage
  - Quick-Start Tutorial
- 
- **Community Edition (OSS)** offers the following features:
    - Foundation of all Workbench editions
    - Full-featured, powerful database management tool
    - Open-source GPL (GNU General Public License) available at no charge from our website
  - **Standard Edition (SE)** offers the following features:
    - Commercial extension of OSS version
    - Advanced features
    - Available for purchase from the website
  - **Installation**
    - Requirements for installation are listed in the *MySQL Workbench Reference Manual*: <http://dev.mysql.com/doc/workbench/en/index.html>.
    - MySQL Workbench download webpage:  
<http://dev.mysql.com/downloads/workbench/>

# MySQL Proxy

- Program that manages communication between your client and MySQL servers:
  - Monitors
  - Analyzes
  - Transforms
- Enables a wide variety of uses:
  - Load balancing
  - Failover
  - Query analysis
  - Query filtering
  - Query modification

MySQL Proxy connects over the network by using the MySQL network protocol and provides communication between one or more MySQL servers and one or more MySQL clients. In the most basic MySQL Proxy configuration, you can:

- Simply pass queries from the clients to the MySQL server and back
- Use it without modification with any MySQL-compatible client that uses the protocol. This includes the `mysql` command-line client, any clients that uses the MySQL client libraries, and any connector that supports the MySQL network protocol.

In addition to the basic pass-through configuration, you can also use MySQL Proxy to:

- Monitor and alter the communication between the client and the server
- Use query interception to add profiling, insert additional queries, and remove the additional results. Interception of the exchanges is scriptable.

The proxy enables you to perform additional monitoring, filtering, or manipulation of queries without requiring you to make any modifications to the client and without the client even being aware that it is communicating with anything but a genuine MySQL server.

MySQL Proxy is still under development and has not been released as GA software.

For more information about MySQL Proxy, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/mysql-proxy.html>.

# MySQL Connectors

- Application programming interfaces (APIs)
  - Also known as “database drivers”
- Compatible with industry standards ODBC and JDBC
- Available for Linux and Windows
- Officially MySQL-supported connectors:
  - Connector/ODBC
  - Connector/J
  - Connector/.NET
  - Connector/Python
  - Connector/C and Connector/C++
  - OpenOffice
- Multiple PHP extensions and APIs are supported.

Any system that works with ODBC or JDBC can use MySQL.

- **Connector/ODBC:** Standardized database driver for Linux, Windows, Mac OS X, and UNIX platforms
- **Connector/J:** Standardized database driver for Java platforms and development
- **Connector/.Net:** Standardized database driver for .NET platforms and development
- **Connector/Python:** Standardized database driver for Python applications
- **Connector/C++:** Standardized database driver for C++ development
- **Connector/C (`libmysql`):** Client library for C development
- **MySQL Connector for OpenOffice:** Driver for OpenOffice.org 3.1 (or later releases)

Multiple PHP extensions and APIs are supported:

- **mysql** extension (uses `libmysql`)
- **mysqli** extension (can use `libmysql` or `mysqlnd`)
- **mysql** support for PDO (can use `libmysql` or `mysqlnd`)
- **mysqlnd:** The MySQL native driver for PHP is an additional, alternative way to connect from PHP 5.3 (or later) to the MySQL Server 4.1 (or later).

## Third-Party APIs

- For several programming languages
- Third-party and community supported
- Many are based on the C client library **libmysql**.
- Some are native implementations of the client protocol.
- Some languages:
  - C++
  - Eiffel
  - Perl
  - Python
  - Ruby
  - TCL

Most of the third-party APIs are based on the C client library and provide a binding for some other language. Some natively implement the `mysql` client protocol.

Although members of the MySQL development team often work closely with the developers of these products, the APIs do not receive official support from Oracle. Contact the developers to determine whether future support is available for APIs that are not currently supported by Oracle.

The biggest concern with migrating APIs is to ensure that the request you send to the server (through the API) is formatted correctly for the server version that is being used.

## **Quiz**

The \_\_\_\_\_ GUI tool can be used to create a database model.

- a. MySQL Proxy
- b. MySQL Enterprise Monitor
- c. MySQL Workbench
- d. MySQL Connectors

# **Chapter 3**

## **Obtaining Metadata**

# **Objectives**

After completing this lesson, you should be able to:

- List the available metadata access methods
- Recognize the structure of the INFORMATION\_SCHEMA database (schema)
- Use the available commands to view metadata
- Describe the differences between SHOW statements and the INFORMATION\_SCHEMA tables
- Use the mysqlshow client program
- Use the INFORMATION\_SCHEMA tables to create shell commands and SQL statements

# Metadata Access Methods

- View data that describes the structure of the database.
- Query the **INFORMATION\_SCHEMA** database tables.
  - They contain data about all objects managed by the MySQL database server.
- Use **SHOW** statements.
  - MySQL-proprietary statements for obtaining database and table information
- Use the **DESCRIBE** (or **DESC**) statement.
  - A shortcut to inspect table structure and column properties
- Use the **mysqlshow** client.
  - A command-line interface to the **SHOW** syntax

A database is a structured collection of data. Metadata is “data about data.” Using the following methods, MySQL provides access to metadata for databases, tables, and other objects managed by the database server:

- **INFORMATION\_SCHEMA:** The MySQL server contains a data dictionary implemented as a database (schema) named INFORMATION\_SCHEMA that includes a number of objects that appear to be tables.
- **SHOW statements:** Proprietary syntax to obtain data on server statistics, schemas, and schema objects:
  - **SHOW DATABASES** and **SHOW TABLES:** Return lists of database and table names
  - **SHOW COLUMNS:** Produces definitions of columns in a table
  - The **SELECT** privilege is required for use of SHOW statements.
- **DESCRIBE:** A SQL statement shortcut you can use to inspect table structure and column properties
- **mysqlshow:** The client program that you can use as a command-line front end to a few of the SHOW statements. The arguments you set determine the information to display, and then the program issues the appropriate **SHOW** statement and displays the results of the statement.

## **INFORMATION\_SCHEMA Database**

- Serves as a central repository for database metadata
  - Schema and schema objects
  - Server statistics (status variables, settings, connections)
- Is kept in a table format allowing flexible access
  - Using arbitrary **SELECT** statements
- Is a “virtual database”
  - Tables are not “real” tables (base tables) but are “system views.”
  - Tables are filled dynamically according to the privileges of the current user.

The INFORMATION\_SCHEMA database serves as a central repository for database metadata. It is a “virtual database” in the sense that it is not stored anywhere on disk; however, it contains tables like any other database, and the contents of its tables can be accessed using SELECT like any other tables. Furthermore, SELECT statements can be used to obtain INFORMATION\_SCHEMA tables.

Because it is a virtual database, MySQL builds query results by reading table and other object metadata. When you execute a query that reads information about many tables, MySQL must execute many disk-level operations, which can take a lot of time.

## INFORMATION\_SCHEMA Tables

List all tables in the **INFORMATION\_SCHEMA** database:

| TABLE_NAME                            |
|---------------------------------------|
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ...                                   |
| USER_PRIVILEGES                       |
| VIEWS                                 |

The INFORMATION\_SCHEMA tables contain the following types of information:

### Table Information

- **COLUMNS**: Columns in tables and views
- **ENGINES**: Storage engines
- **SCHEMATA**: Databases
- **TABLES**: Tables in databases
- **VIEWS**: Views in databases

### Partitioning

- **PARTITIONS**: Table partitions
- **FILES**: Files in which MySQL NDB disk data tables are stored

### Privileges

- **COLUMN\_PRIVILEGES**: Column privileges held by MySQL user accounts
- **SCHEMA\_PRIVILEGES**: Database privileges held by MySQL user accounts
- **TABLE\_PRIVILEGES**: Table privileges held by MySQL user accounts
- **USER\_PRIVILEGES**: Global privileges held by MySQL user accounts

## Character Set Support

- **CHARACTER\_SETS**: Available character sets
- **COLLATIONS**: Collations for each character set
- **COLLATION\_CHARACTER\_SET\_APPLICABILITY**: Collations that are applicable to a particular character set

## Constraints and Indexes

- **KEY\_COLUMN\_USAGE**: Constraints on key columns
- **REFERENTIAL\_CONSTRAINTS**: Foreign keys
- **STATISTICS**: Table indexes
- **TABLE\_CONSTRAINTS**: Constraints on tables

## Server Settings and Status

- **KEY\_COLUMN\_USAGE**: Constraints
- **GLOBAL\_STATUS**: The status values for all connections to MySQL
- **GLOBAL\_VARIABLES**: The values that are used for new connections to MySQL
- **PLUGINS**: Server plugins
- **PROCESSLIST**: Indicates which threads are running
- **SESSION\_STATUS**: The status values for the current connection to MySQL
- **SESSION\_VARIABLES**: The values that are in effect for the current connection to MySQL

## Routines and Related Information

- **EVENTS**: Scheduled events
- **ROUTINES**: Stored procedures and functions
- **TRIGGERS**: Triggers in databases
- **PARAMETERS**: Stored procedure and function parameters, and stored functions

## InnoDB

- **INNODB\_CMP** and **INNODB\_CMP\_RESET**: Status on operations related to compressed InnoDB tables
- **INNODB\_CMPMEM** and **INNODB\_CMPMEM\_RESET**: Status on compressed pages within the InnoDB buffer pool
- **INNODB\_LOCKS**: Each lock that an InnoDB transaction has requested and holds
- **INNODB\_LOCK\_WAITS**: One or more row locks for each blocked InnoDB transaction
- **INNODB\_TRX**: Every transaction currently executing inside InnoDB
- **TABLESPACES**: Active tablespaces

For more information about the `INFORMATION_SCHEMA` tables, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/information-schema.html>.

## **INFORMATION\_SCHEMA Table Columns**

List **INFORMATION\_SCHEMA** database table columns:

```
mysql> SELECT COLUMN_NAME
      -> FROM INFORMATION_SCHEMA.COLUMNS
      -> WHERE TABLE_SCHEMA = 'INFORMATION_SCHEMA'
      -> AND TABLE_NAME = 'VIEWS';
+-----+
| COLUMN_NAME |
+-----+
| TABLE_CATALOG |
| TABLE_SCHEMA |
| TABLE_NAME |
| VIEW_DEFINITION |
| CHECK_OPTION |
| IS_UPDATABLE |
| DEFINER |
| SECURITY_TYPE |
| CHARACTER_SET_CLIENT |
| COLLATION_CONNECTION |
+-----+
```

## Using SELECT with INFORMATION\_SCHEMA

- Specify which table and columns to retrieve.
- Retrieve only specific conditions by using a **WHERE** clause.
- Group or sort results.
- Use **JOIN**, **UNION**, and subqueries.
- Retrieve results into another table.
- Create views on top of **INFORMATION\_SCHEMA** tables.

When you retrieve metadata from the **INFORMATION\_SCHEMA** tables by using **SELECT** statements, you can use any of the usual **SELECT** features.

You can retrieve the result of an **INFORMATION\_SCHEMA** query into another table by using the **CREATE TABLE . . . SELECT** statement or the **INSERT . . . SELECT** statement. This enables you to save the results and use them later in other statements.

## INFORMATION\_SCHEMA: Examples

```
mysql> SELECT TABLE_NAME, ENGINE  
-> FROM INFORMATION_SCHEMA.TABLES  
-> WHERE TABLE_SCHEMA = 'world_innodb'; 1  
  
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME  
-> FROM INFORMATION_SCHEMA.COLUMNS  
-> WHERE DATA_TYPE = 'set'; 2  
  
mysql> SELECT CHARACTER_SET_NAME, COLLATION_NAME  
-> FROM INFORMATION_SCHEMA.COLLATIONS  
-> WHERE IS_DEFAULT = 'Yes'; 3  
  
mysql> SELECT TABLE_SCHEMA, COUNT(*)  
-> FROM INFORMATION_SCHEMA.TABLES  
-> GROUP BY TABLE_SCHEMA; 4  
  
mysql> DELETE FROM INFORMATION_SCHEMA.VIEWS;  
ERROR 1044 (42000): Access denied for user  
'root'@'localhost' to database 'information_schema' 5
```

The slide examples demonstrate how to exploit various features of SELECT to pull out information in different ways from INFORMATION\_SCHEMA tables:

1. Displays the storage engines used for the tables in a given database
2. Finds all the tables that contain SET columns
3. Displays the default collation for each character set
4. Displays the number of tables in each database
5. The INFORMATION\_SCHEMA tables are read-only and cannot be modified with statements such as INSERT, DELETE, or UPDATE. The server produces an error if you execute these types of statements in an attempt to change the data in the INFORMATION\_SCHEMA tables.

## Creating Shell Commands with INFORMATION\_SCHEMA Tables

- Use the INFORMATION\_SCHEMA tables to obtain information about creating shell commands.
- Use **SELECT** with **CONCAT** to create mysqldump scripts:

```
mysql> SELECT CONCAT("mysqldump -uroot -p ",  
-> TABLE_SCHEMA, " ", TABLE_NAME, " >> ",  
-> TABLE_SCHEMA,".bak.sql")  
-> FROM TABLES WHERE TABLE_NAME LIKE 'Country%';
```

- Results in the following shell commands:

```
shell> mysqldump -uroot -p world_innodb Country  
      >> world_innodb.bak.sql  
  
shell> mysqldump -uroot -p world_innodb  
      CountryLanguage >> world_innodb.bak.sql
```

- Place the result in an output file by adding:

```
...INTO OUTFILE '/Country_Dump.sh';
```

Using the CONCAT function, you can combine string content to create shell scripts that can be executed from the command line. As shown in the slide example, the SQL statement produces an output that dumps only those tables from the `world_innodb` database that begin with the word “Country.”

The output produces a shell script that can execute properly on a shell command line. The next step would be to store this output in a batch file that could be executed from a shell command line. This is done by adding the clause `INTO OUTFILE`:

```
mysql> SELECT CONCAT("mysqldump -uroot -p ",  
-> TABLE_SCHEMA, " ",TABLE_NAME, " >> ",TABLE_SCHEMA,".sql")  
-> FROM TABLES WHERE TABLE_NAME LIKE 'Country%'  
-> INTO OUTFILE '/Country_Dump.sh';
```

This file could then be executed from the command line, which runs the two `mysqldump` commands shown in the slide:

```
shell> \tmp\Country_Dump.sh  
shell> \tmp\mysqldump -uroot -poracle world_innodb Country >>  
      world_innodb.sql  
shell> \tmp\mysqldump -uroot -poracle world_innodb Country_Language >>  
      world_innodb.sql
```

## Creating SQL Statements with INFORMATION\_SCHEMA Tables

Use the mysql command to create SQL statements.

- Use the **-e** option to enter a **SELECT/CONCAT** statement:

```
shell> mysql -uroot -p --silent --skip-column-names -e
  "SELECT CONCAT('CREATE TABLE ', TABLE_SCHEMA, '.',
    TABLE_NAME, '_backup LIKE ', TABLE_SCHEMA, '.',
    TABLE_NAME, ';') FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'world_innodb';"
```

- Results in the following statements sent to standard output:

```
CREATE TABLE world_innodb.City_backup LIKE
  world_innodb.City;
CREATE TABLE world_innodb.Country_backup LIKE
  world_innodb.Country_backup;
CREATE TABLE world_innodb.CountryLanguage_backup LIKE
  world_innodb.CountryLanguage_backup;
```

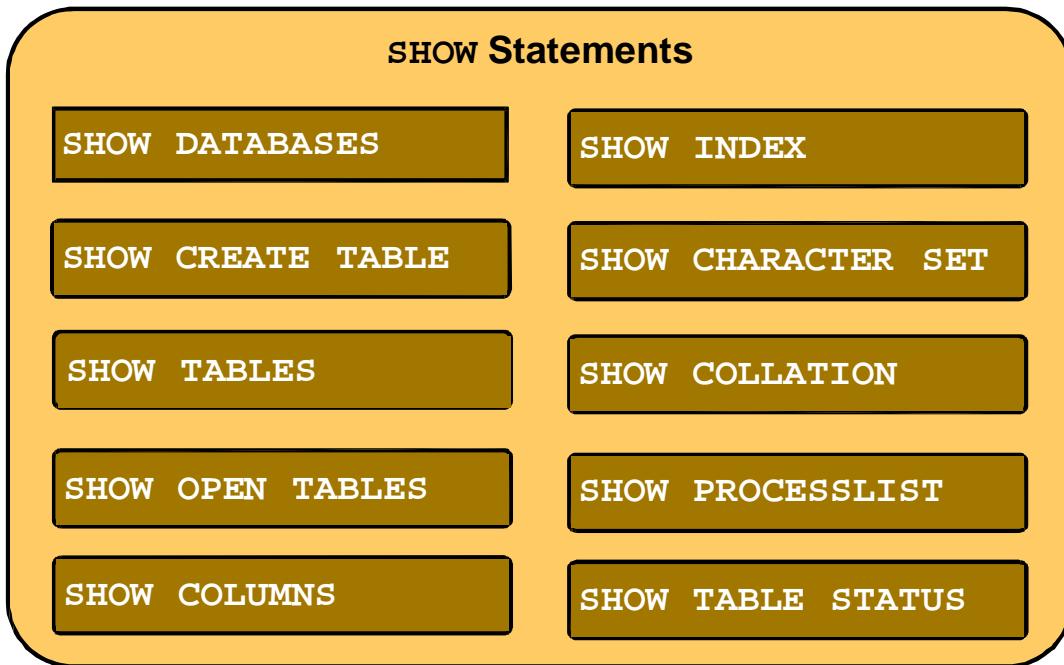
The INFORMATION\_SCHEMA tables create SQL statements that you can execute from the command line. The example in the slide uses the mysql command to execute a statement that makes an exact copy of all the tables in the world\_innodb database. This command creates a SQL output that, if executed, would create three backup tables based on the tables in the world\_innodb database.

**Note:** The **--silent** command removes the column heading from the output, and the **--skip-column-names** command removes the formatting around the output (the formatting that makes the output look like a table). These two commands are used to ensure that the commands themselves are interpreted properly without any problems with external formatting or header rows interfering with execution.

Adding the pipe symbol ( | ) followed by the execution of the mysql command sends these SQL statements to the MySQL server to be executed:

```
shell> mysql -uroot -p --silent --skip-column-names -e
  "SELECT CONCAT('CREATE TABLE ', TABLE_SCHEMA, '.',
    TABLE_NAME, '_backup LIKE ', TABLE_SCHEMA, '.',
    TABLE_NAME, ';') FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'world_innodb';" | mysql -uroot -poracle
```

# MySQL-Supported SHOW Statements



In addition to INFORMATION\_SCHEMA tables, MySQL also supports the SHOW and DESCRIBE statements as alternative means of accessing metadata. The SHOW and DESCRIBE syntax are not as flexible as using INFORMATION\_SCHEMA queries, but for many purposes the SHOW and DESCRIBE syntax is sufficient. In those cases, it is often quicker and easier for you to use this MySQL specific syntax.

The SHOW statement can be used in many forms, as follows:

- **SHOW DATABASES:** Lists the names of the available databases
- **SHOW TABLES:** Lists the tables in the default database
- **SHOW TABLES FROM <database\_name>:** Lists the tables in the specified database
- **SHOW COLUMNS FROM <table\_name>:** Displays column structure for the table
- **SHOW INDEX FROM <table\_name>:** Displays information about the indexes and index columns in the table
- **SHOW CHARACTER SET:** Displays the available character sets along with their default collations
- **SHOW COLLATION:** Displays the collations for each character set

## SHOW Statement: Examples

Some commonly used **SHOW** statements:

```
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql          |
| test           |
| world          |
| innodb         |
+-----+  
  
mysql> SHOW TABLES;  
mysql> SHOW TABLES FROM mysql;  
mysql> SHOW TABLES FROM INFORMATION_SCHEMA;  
mysql> SHOW COLUMNS FROM CountryLanguage;  
mysql> SHOW FULL COLUMNS FROM CountryLanguage\G
```

## Additional SHOW Statement Examples

- **SHOW** with **LIKE** and **WHERE**:

```
mysql> SHOW DATABASES LIKE 'm%';  
  
mysql> SHOW COLUMNS FROM Country  
      -> WHERE `Default` IS NULL;
```

- Other **SHOW** statements:

```
mysql> SHOW INDEX FROM City\G  
  
mysql> SHOW CHARACTER SET;  
  
mysql> SHOW COLLATION;
```

## DESCRIBE Statement

- Equivalent to **SHOW COLUMNS**
- General syntax:

```
mysql> DESCRIBE <table_name>;
```

- Shows INFORMATION\_SCHEMA table information

```
mysql> DESCRIBE INFORMATION_SCHEMA.CHARACTER_SETS;
```

| Field                | Type        | Null | Key | Default | Extra |
|----------------------|-------------|------|-----|---------|-------|
| CHARACTER_SET_NAME   | varchar(64) | NO   |     |         |       |
| DEFAULT_COLLATE_NAME | varchar(64) | NO   |     |         |       |
| DESCRIPTION          | varchar(60) | NO   |     |         |       |
| MAXLEN               | bigint(3)   | NO   |     | 0       |       |

### DESCRIBE

DESCRIBE can be abbreviated as DESC, as follows:

```
mysql> DESCRIBE table_name;  
and  
mysql> DESC table_name;
```

The following is equivalent to the above DESCRIBE/DESC examples:

```
mysql> SHOW COLUMNS FROM table_name;
```

However, whereas SHOW COLUMNS supports an optional LIKE and WHERE clause, DESCRIBE does not.

### EXPLAIN

EXPLAIN is equivalent to DESCRIBE when given a table name as its parameter:

```
mysql> EXPLAIN table_name;
```

EXPLAIN is described in further detail in the lesson titled “Introduction to Performance Tuning.”

## mysqlshow Client

- Information about structure of databases and tables
  - Similar to **SHOW** statements
- General syntax:

```
shell> mysqlshow [options] [db_name  
[table_name [column_name]]]
```

- Options can be standard connection parameters.

The `mysqlshow` client provides a command-line interface to various forms of the `SHOW` statement that list the names of databases, tables within a database, or information about table columns or indexes.

The options part of the `mysqlshow` client can include any of the standard connection parameter options, such as `--host` or `--user`.

You must supply options if the default connection parameters are not appropriate. `mysqlshow` also accepts options that are specific to its own operation.

Invoke `mysqlshow` with the `--help` option to see a complete list of its options.

The action performed by `mysqlshow` depends on the number of non-option arguments that are provided.

## mysqlshow: Examples

Show information for all databases, or for a specific database, table, and/or column:

```
shell> mysqlshow -u<user_name> -p<password> ①
+-----+
| Databases |
+-----+
| information_schema |
| mysql |
| performance_schema |
| test |
| world innodb |
+-----+  
  
shell> mysqlshow world_innodb ②
shell> mysqlshow world_innodb City ③
shell> mysqlshow world_innodb City CountryCode ④
shell> mysqlshow "w%" ⑤
```

The slide examples demonstrate some uses of the mysqlshow client:

1. With no arguments, mysqlshow displays a result similar to that of SHOW DATABASES.
2. With a single argument, mysqlshow interprets it as a database name and displays a result similar to that of SHOW TABLES for the database.
3. With two arguments, mysqlshow interprets them as a database and table name and displays a result similar to that of SHOW FULL COLUMNS for the table.
4. With three arguments, the output is the same as for two arguments except that mysqlshow takes the third argument as a column name and displays SHOW FULL COLUMNS output only for that column.
5. If the final argument on the command line contains special characters, mysqlshow interprets the argument as a pattern and displays only the names that match the pattern. The special characters are % or \* to match any sequence of characters, and \_ or ? to match any single character. This example command shows only those databases with a name that begins with w.

**Note:** These examples require the use of user and password parameters as part of command execution.

## **Summary**

In this lesson, you should have learned how to:

- List the available metadata access methods
- Recognize the structure of the INFORMATION\_SCHEMA database (schema)
- Use the available commands to view metadata
- Describe the differences between SHOW statements and the INFORMATION\_SCHEMA tables
- Use the mysqlshow client program
- Use the INFORMATION\_SCHEMA tables to create shell commands and SQL statements

## **Chapter 4**

### **Transaction, Locking and Innodb Storage Engine**

## **Objectives**

After completing this lesson, you should be able to:

- Use transaction control statements to run multiple SQL statements concurrently
- Explain the ACID properties
- Describe the transaction isolation levels
- Use locking to protect transactions

## Transactions

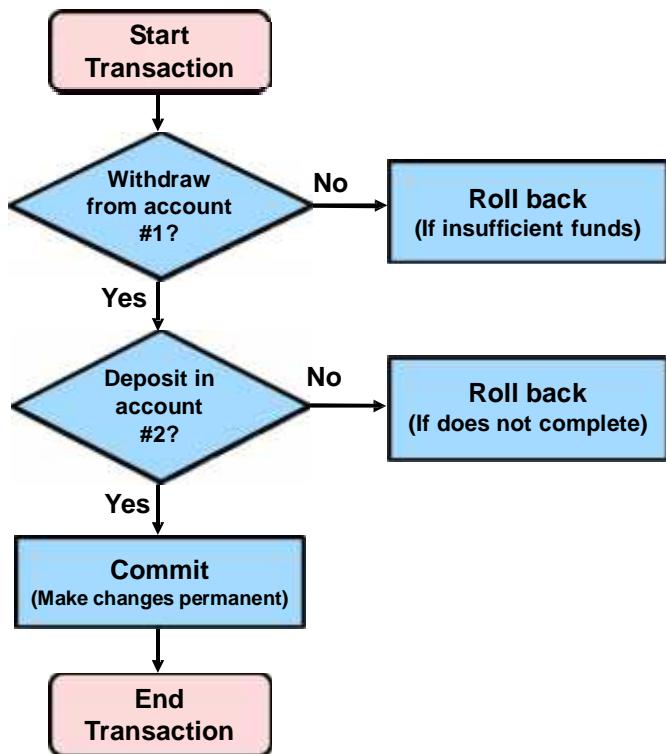
- A collection of data manipulation execution steps that are treated as a single unit of work
  - Use to group multiple statements.
  - Use when multiple clients are accessing data from the same table concurrently.
- All or none of the steps succeed.
  - Execute if all steps are good.
  - Cancel if steps have error or are incomplete.
- ACID compliant

A transaction is a way for you to execute one or more SQL statements as a single unit of work, so that either all or none of the statements succeed. This appears to happen in isolation of work being done by any other transactions. If all the statements succeed, you commit the transaction to record their effect permanently in the database. If an error occurs during the transaction, you roll back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was in prior to the point at which the transaction began.

**Note:** In MySQL, transactions are supported only for those tables that use a transactional storage engine (like InnoDB). These statements have no noticeable effect on tables managed by a non-transactional storage engine.

# Transaction Diagram

Example:  
A banking transaction



The diagram in the slide is an example of an attempted transfer of \$1,000 from your savings account to your checking account. You would not be happy if the money was successfully withdrawn from your savings account but was never deposited into your checking account.

To protect against this kind of error, the program that handles your transfer request would first begin a transaction and then issue the SQL statements needed to move the money from your savings to your checking account. It ends the transaction only if everything succeeds. If a problem occurs, the program would instruct the server to undo all the changes made since the transaction began.

# ACID

- **Atomic**
  - All statements execute successfully or are canceled as a unit.
- **Consistent**
  - A database that is in a consistent state when a transaction begins is left in a consistent state by the transaction.
- **Isolated**
  - One transaction does not affect another.
- **Durable**
  - All changes made by transactions that complete successfully are recorded properly in the database.
    - Changes are not lost.

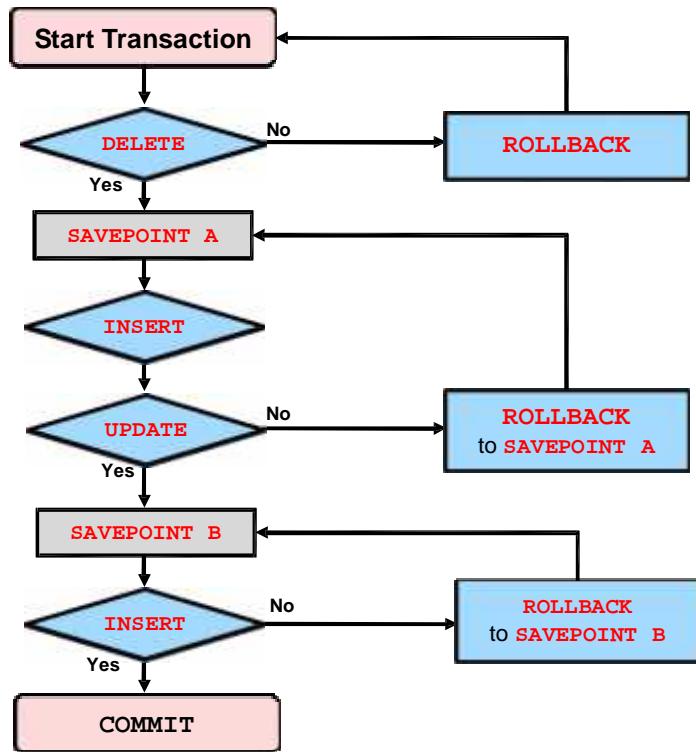
Transactional processing provides stronger guarantees about the outcome of the database operations, but also requires more overhead in CPU cycles, memory, and disk space. Transactional properties are essential for some applications but not for others, and you can choose which ones make the most sense for your applications.

Financial operations typically need transactions, and the guarantees of data integrity outweigh the cost of additional overhead. On the other hand, for an application that logs webpage accesses to a database table, a loss of a few records if the server host crashes might be tolerable.

# Transaction SQL Control Statements

- **START TRANSACTION** (or **BEGIN**): Explicitly begins a new transaction
  - **SAVEPOINT**: Assigns a location in the process of a transaction for future reference
  - **COMMIT**: Makes the changes from the current transaction permanent
  - **ROLLBACK**: Cancels the changes from the current transaction
  - **ROLLBACK TO SAVEPOINT**: Cancels the changes executed after the savepoint
  - **RELEASE SAVEPOINT**: Removes the savepoint identifier
  - **SET AUTOCOMMIT**: Disables or enables the default autocommit mode for the current connection
- 
- **START TRANSACTION** or **BEGIN**: The transaction is open until it is explicitly closed by COMMIT or ROLLBACK.
  - **SAVEPOINT**: A unique identifier for a position that you can roll back to, within a transaction
  - **ROLLBACK TO SAVEPOINT**: The savepoint continues to exist after the rollback, allowing it to be “reused.”
  - **RELEASE SAVEPOINT**: Does not remove any of the transactional statements

# SQL Control Statements Flow: Example



The flow shown in the slide assumes that the AUTOCOMMIT mode is enabled. The transaction is automatically committed if the statement executes successfully, permanently saving any changes caused by executing the statement. If the statement does not execute successfully, the transaction is automatically rolled back, undoing any changes resulting from executing the statement.

**Note:** AUTOCOMMIT mode is explained in detail in the next slide.

## AUTOCOMMIT Mode

- Determines how and when new transactions are started
- AUTOCOMMIT mode is enabled by default:
  - Implicitly commits each statement as a transaction
  - Override for a single transaction with **START TRANSACTION**
- Set **AUTOCOMMIT** mode to 0 in an option file, or:

```
SET GLOBAL AUTOCOMMIT=0;
SET SESSION AUTOCOMMIT=0;
SET @@AUTOCOMMIT :=0;
```

- When **AUTOCOMMIT** is disabled, transactions span multiple statements by default.
  - You can end a transaction with **COMMIT** or **ROLLBACK**.
- Check the **AUTOCOMMIT** setting with **SELECT**:

```
SELECT @@AUTOCOMMIT;
```

AUTOCOMMIT mode determines how and when new transactions are started. When AUTOCOMMIT mode is enabled, each SQL statement implicitly starts a new transaction. When each statement completes, the transaction is committed. The result of this is that when AUTOCOMMIT mode is enabled, changes to transactional tables are effective immediately.

You can override this behavior by explicitly starting a new transaction with **START TRANSACTION**. Subsequent statements are not persisted to the tables until you end the transaction with an explicit **COMMIT**.

When you enable AUTOCOMMIT you do not disable transactions. For example, when you execute a single statement that affects multiple rows and one modification fails, the outcome differs when you use a transactional storage engine such as InnoDB, and a nontransactional engine such as MyISAM. With a MyISAM table, the statement terminates as soon as the error is encountered, leaving the rows that have already been inserted in the table. With an InnoDB table, all rows that have already been inserted are withdrawn from the table and the operation therefore has no net effect.

By default, AUTOCOMMIT is enabled globally. Disable AUTOCOMMIT globally within an option file, or disable it for a specific session by setting the **autocommit** variable.

# Implicit Commit

- An implicit commit terminates the current transaction.
- SQL statements that implicitly commit:
  - `START TRANSACTION`
  - `SET AUTOCOMMIT = 1`
- Non-transactional statements that cause a commit:
  - Data definition statements (`ALTER`, `CREATE`, `DROP`)
  - Administrative statements (`GRANT`, `REVOKE`, `SET PASSWORD`)
  - Locking statements (`LOCK TABLES`, `UNLOCK TABLES`)
- Example of statements that cause an implicit commit:
  - `TRUNCATE TABLE`
  - `LOAD DATA INFILE`

The `COMMIT` statement always *explicitly* commits the current transaction. Other transaction control statements (such as those listed in the slide) also have the effect of *implicitly* committing the current transaction.

Apart from these transaction control statements, other types of statements may have the effect of implicitly committing (and thereby terminating) the current transaction. These statements behave as if you issued a `COMMIT` prior to executing the actual statement. In addition, these statements are themselves non-transactional, which means that they cannot be rolled back if they succeed.

In general, the data definition statements, data access and user management (administrative) statements, and locking statements have this effect.

**Note:** There are a number of exceptions, and not all of these statements cause an implicit commit in all versions of the server. However, it is advisable to treat all non-DML statements as if they cause an implicit commit.

For a full list of statements that cause implicit commits, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/implicit-commit.html>.

# Transactional Storage Engines

List the engine characteristics with **SHOW ENGINES**:

```
mysql> SHOW ENGINES\G
***** 2. row *****
Engine: InnoDB
Support: DEFAULT
Comment: Supports transactions, row-level locking,
and foreign keys
Transactions: YES
XA: YES
Savepoints: YES
***** 1. row *****
Engine: MyISAM
Support: YES
Comment: MyISAM storage engine
Transactions: NO
XA: NO
Savepoints: NO
...
```

To ensure that a transactional storage engine is compiled into your MySQL server and that it is available at run time, use the **SHOW ENGINES** statement.

The value in the **Support** column is **YES** or **NO** to indicate that the engine is or is not available. If the value is **DISABLED**, the engine is present but turned off. The value **DEFAULT** indicates the storage engine that the server uses by default. The engine designated as **DEFAULT** should be considered available. The **Transactions**, **XA**, and **Savepoints** columns indicate whether the storage engine supports those features.

# Transaction Isolation Problems

Three common problems:

- “Dirty” read
  - When a transaction reads the changes made by another uncommitted transaction
- Non-repeatable read
  - When changes from another committed transaction cause a prior read operation to be non-repeatable
- Phantom read (or *phantom row*)
  - A row that appears but was not previously visible within the same transaction

Multiple pending transactions may exist simultaneously within the server (at most one transaction per session). When multiple clients are accessing data from the same table concurrently, the following consistency issues can occur:

- **Dirty read:** Suppose that transaction T1 modifies a row. If transaction T2 reads the row and sees the modification even though T1 has not committed it, that is a dirty read. One reason this is a problem is that if T1 rolls back, the change is undone but T2 does not know that.
- **Non-repeatable read:** The same read operation yields different results when it is repeated at a later time within the same transaction.
- **Phantom read:** Suppose that transactions T1 and T2 begin, and T1 reads some rows. If T2 inserts a new row and T1 sees that row when it repeats the same read operation, a phantom read has occurred (the new row being the phantom row).

# Isolation Levels

Four isolation levels:

- **READ UNCOMMITTED**
  - Allows a transaction to see uncommitted changes made by other transactions
- **READ COMMITTED**
  - Allows a transaction to see committed changes made by other transactions
- **REPEATABLE READ**
  - Ensures consistent **SELECT** output for each transaction
  - Default level for InnoDB
- **SERIALIZABLE**
  - Completely isolates the effects of a transaction from others

If one client's transaction changes data, should transactions for other clients see those changes or should they be isolated from them? The transaction isolation level determines the ways in which simultaneous transactions interact when accessing the same data.

Use storage engines to implement isolation levels. Isolation level choices vary among database servers, so the levels as implemented by InnoDB might not correspond exactly to levels as implemented in other database systems.

InnoDB implements four isolation levels that control the extent to which changes made by transactions are noticeable to other simultaneously occurring transactions:

- **READ UNCOMMITTED**: Allows dirty reads, non-repeatable reads, and phantom reads to occur
- **READ COMMITTED**: Allows non-repeatable reads and phantoms to occur. Uncommitted changes remain invisible.
- **REPEATABLE READ**: Gets the same result both times, regardless of committed or uncommitted changes made by other transactions. In other words, it gets a consistent result from different transactions on the same data.
- **SERIALIZABLE**: Similar to **REPEATABLE READ** with the additional restriction that rows selected by one transaction cannot be changed by another until the first transaction finishes

# Isolation Level Problems

| Isolation Level  | Dirty Read   | Non-Repeatable Read | Phantom Read |
|------------------|--------------|---------------------|--------------|
| Read Uncommitted | Possible     | Possible            | Possible     |
| Read Committed   | Not possible | Possible            | Possible     |
| Repeatable Read  | Not possible | Not possible        | Possible*    |
| Serializable     | Not possible | Not possible        | Not possible |

\* Not possible for InnoDB, which uses snapshots for Repeatable Read

# Setting the Isolation Level

- Set the level at server startup.
  - Use the `--transaction-isolation` option with the `mysqld` command.
  - Or set `transaction-isolation` in the configuration file:

```
[mysqld]
transaction-isolation = <isolation_level>
```

- Set for a running server by using a `SET TRANSACTION ISOLATION LEVEL` statement.
  - Syntax examples:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL <isolation_level>;
SET SESSION TRANSACTION ISOLATION LEVEL <isolation_level>;
SET TRANSACTION ISOLATION LEVEL <isolation_level>;
```

Set the `<isolation_level>` value in the option file or on the command line to READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, or SERIALIZABLE.

For the `SET TRANSACTION ISOLATION LEVEL` statement, set the `<isolation_level>` value to READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

The transaction level can be set either at the global or at the session level. Without an explicit specification, the transaction isolation level is set at the session level. For example, the following statement sets the isolation level to READ COMMITTED for the current `mysql` session:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

This is equivalent to:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

To set the default level for all subsequent `mysql` connections, use the `GLOBAL` keyword instead of `SESSION`:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

**Note:** Setting the default transaction isolation level globally applies to all new client connections established from that point on. Existing connections are unaffected.

## Global Isolation Level

Requires the SUPER privilege:

```
mysql> SELECT @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+


mysql> SELECT @@global.tx_isolation,
        @@session.tx_isolation;
+-----+-----+
| @@global.tx_isolation | @@session.tx_isolation |
+-----+-----+
| READ-UNCOMMITTED      | REPEATABLE-READ   |
+-----+-----+
```

A client can always modify the transaction isolation level for its own session, but changing the default transaction isolation level globally requires the SUPER privilege.

To find out what the current isolation level is, use the `tx_isolation` server variable.

When this variable is used unprefixed, the session transaction isolation level is returned. Use the `global` and `session` prefixes to explicitly obtain the global or session isolation level, respectively.

You can also use the server variable to set the transaction isolation level. The same isolation levels are valid as in the `SET TRANSACTION ISOLATION LEVEL` syntax, except that it must be a string representation (rather than plain keywords) and you must separate the words that define an isolation level by a single hyphen (dash or minus sign) rather than whitespace.

## Transaction Example: Isolation

| Session 1  | Session 2  |
|--|--|
| <pre>mysql&gt; PROMPT s1&gt;  s1&gt; SET GLOBAL TRANSACTION -&gt; ISOLATION LEVEL READ COMMITTED;  s1&gt; SELECT @@global.tx_isolation; +-----+   @@global.tx_isolation   +-----+   READ-COMMITTED   +-----+</pre> |  |
|  | <pre>mysql&gt; PROMPT s2&gt;  s2&gt; START TRANSACTION;  s2&gt; INSERT INTO City -&gt; (Name, CountryCode,Population) -&gt; VALUES ('Sakila', 'SWE', 1);</pre> |
| <pre>s1&gt; SELECT Name, CountryCode -&gt; FROM City -&gt; WHERE Name = 'Sakila'; Empty Set (0.0 sec)</pre>  |  |

The procedure in the slide shows the effect of transaction isolation for two transactions occurring in two different simultaneous mysql sessions. The first SELECT in Session 1 gives an empty set, because the current isolation level prevents this transaction from seeing uncommitted changes.

## Transaction Example: Isolation

| Session 1  | Session 2   |
|--|-------------|
|  | s2> COMMIT; |
| <pre>s1&gt; SELECT Name, CountryCode -&gt; FROM City -&gt; WHERE Name = 'Sakila'; +-----+   Name      CountryCode   +-----+   Sakila   SWE +-----+</pre> |             |

This example assumes that all tables are using the InnoDB storage engine, row-based logging is taking place, and AUTOCOMMIT is enabled for both sessions.

# Locking Concepts

- MySQL uses a multithreaded architecture.
  - Problems arise with multiple client access to a table.
  - Client coordination is necessary.
- Locking is a mechanism to prevent concurrency problems.
  - Managed by server
  - Locks for one client (to restrict others)
- Types of locks:
  - Shared lock
  - Exclusive lock

Locking is a mechanism that prevents problems from occurring with simultaneous data access by multiple clients. It places a lock on data on behalf of one client to restrict access by other clients to the data until the lock has been released. The lock allows access to data by the client that holds the lock, but places limitations on what operations can be done by other clients that are contending for access. The effect of the locking mechanism is to serialize access to data so that when multiple clients want to perform conflicting operations, each must wait its turn. Not all types of concurrent access produce conflicts, so the type of locking that is necessary to allow a client to access data depends on whether the client wants to read or write:

- If a client wants to read data, other clients that want to read the same data do not produce a conflict, and they all can read at the same time. However, another client that wants to write (modify) data must wait until the read has finished.
- If a client wants to write data, all other clients must wait until the write has finished, regardless of whether those clients want to read or write.

A reader must block writers but must not block other readers. A writer must block both readers and writers. Read locks and write locks allow these restrictions to be enforced. Locking makes clients wait for access until it is safe for them to proceed. In this way, locks prevent data corruption by disallowing concurrent conflicting changes and reading of data that is in the process of being changed.

# Explicit Row Locks

InnoDB supports two types of row locking:

- **LOCK IN SHARE MODE**
  - Locks each row with a shared lock

```
SELECT * FROM Country WHERE Code='AUS'  
LOCK IN SHARE MODE\G
```

- **FOR UPDATE**
  - Locks each row with an exclusive lock

```
SELECT counter_field INTO @@counter_field  
FROM child_codes FOR UPDATE;  
  
UPDATE child_codes SET counter_field =  
@@counter_field + 1;
```

InnoDB supports two locking modifiers that may be added to the end of SELECT statements:

- **LOCK IN SHARE MODE clause:** A shared lock, which means that no other transactions can take exclusive locks but other transactions can also use shared locks. Because normal reads do not lock anything, they are not affected by the locks.
- **FOR UPDATE clause:** Locks each selected row with an exclusive lock, preventing others from acquiring any lock on the rows but allowing reading of the rows.

In the REPEATABLE READ isolation level, LOCK IN SHARE MODE can be added to SELECT operations to force other transactions to wait for the transaction if they want to modify the selected rows. This is similar to operating at the SERIALIZABLE isolation level, for which InnoDB implicitly adds LOCK IN SHARE MODE to SELECT statements that have no explicit locking modifier. If it selects rows that have been modified in an uncommitted transaction, it locks the SELECT until that transaction commits.

Deadlocks occur when multiple transactions each require data that another has already locked exclusively.

- When a cyclical dependency occurs between two or more transactions
  - For example, T1 waits for a resource locked by T2, which waits for a resource locked by T3, which waits for a resource locked by T1.
- InnoDB detects and aborts (rollback) one of the transactions and allows the other one to complete.

Deadlocks are a classic problem in transactional databases, but they are not dangerous unless they are so frequent that you cannot run certain transactions at all. A deadlock can occur when:

- Transactions acquire locks on multiple tables, but in the opposite order
- Statements such as UPDATE or SELECT . . . FOR UPDATE lock ranges of index records and gaps, with each transaction acquiring some locks but not others due to a timing issue
- There are multiple transactions, and one is waiting for the other(s) to complete to form a cycle. For example, T1 is waiting for T2, T2 is waiting for T3, and T3 is waiting for T1.

## **Rollback**

When InnoDB performs a complete rollback of a transaction, all locks set by the transaction are released. However, if just a single SQL statement is rolled back as a result of an error, some of the locks set by the statement may be preserved. This happens because InnoDB stores row locks in a format such that it cannot know afterward which lock was set by which statement.

If a SELECT statement calls a stored function in a transaction, and a statement within the function fails, that statement rolls back. Also, if you perform a ROLLBACK after that, the entire transaction rolls back.

## Transaction Example: Deadlock

| Session 1  | Session 2   |
|--|---|
| s1> START TRANSACTION;<br><br>s1> UPDATE Country<br>-> SET Name = 'Sakila'<br>-> WHERE Code = 'SWE'; |   |
|  | s2> START TRANSACTION;<br><br>s2> UPDATE Country<br>-> SET Name = 'World Cup Winner'<br>-> WHERE Code = 'ITA';  |
| s1> DELETE FROM Country<br>-> WHERE Code = 'ITA';  |   |
|  | s2> UPDATE Country<br>-> SET population=1<br>-> WHERE Code = 'SWE';<br>ERROR 1213 (40001): Deadlock<br>found when trying to get lock;<br>try restarting transaction |
| Query OK, 1 row affected (0.0 sec)   |   |

The first DELETE statement hangs while waiting for a lock. During execution of the UPDATE statement, a deadlock is detected in Session 2 due to the conflicts between the two sessions. The UPDATE is aborted, allowing the DELETE from Session 1 to complete.

## Implicit Locks

The MySQL server locks the table (or row) based on the commands issued and the storage engines being used:

| Operation     | InnoDB                   | MyISAM                     |
|---------------|--------------------------|----------------------------|
| SELECT        | No lock*                 | Table-level shared lock    |
| UPDATE/DELETE | Row-level exclusive lock | Table-level exclusive lock |
| ALTER TABLE   | Table-level shared lock  | Table-level shared lock    |

\* No lock unless SERIALIZABLE level, LOCK IN SHARE MODE, or FOR UPDATE is used

InnoDB tables use row-level locking so that multiple sessions and applications can read from and write to the same table simultaneously, without making each other wait and without producing inconsistent results. For this storage engine, avoid using the `LOCK TABLES` statement; it does not offer any extra protection but instead reduces concurrency.

The automatic row-level locking makes these tables suitable for your busiest databases with your most important data, while also simplifying application logic, because you do not need to lock and unlock tables. Consequently, the InnoDB storage engine is the default in MySQL 5.6.

## **Summary**

In this lesson, you should have learned how to:

- Use transaction control statements to run multiple SQL statements concurrently
- Explain the ACID properties
- Describe the transaction isolation levels
- Use locking to protect transactions

## Storage Engines and MySQL

- MySQL provides and maintains several storage engines.
- Each has different characteristics and implications.
- You can choose a specific storage engine when you create a table.

When you create a table, MySQL uses the InnoDB storage engine to create the storage for that table on the hard disk. You can choose an alternative storage engine to use for each table. Typically, you make this choice according to which storage engine offers features that best fit the needs of your application. Each storage engine has a particular set of operational characteristics. These characteristics include the types of locks that are used to manage query contention, and whether the storage engine supports transactions. These engine properties have implications for query processing performance, concurrency, and deadlock prevention.

Although there are many other storage engines available, InnoDB is the best fit for most use cases.

# Available Storage Engines

MySQL provides the following storage engines:

- InnoDB
- MyISAM
- MEMORY
- ARCHIVE
- FEDERATED
- EXAMPLE
- BLACKHOLE
- MERGE
- NDBCLUSTER
- CSV

} *Most common*

Third-party storage engines are also available.

MySQL provides and maintains several storage engines. The MySQL server is also compatible with many third-party storage engines. A MySQL storage engine is a low-level engine inside the database server that takes care of storing and retrieving data, and can be accessed through an internal MySQL API or, in some situations, can be accessed directly by an application. Note that one application can have more than one storage engine in use at any given time. The slide lists the currently supported storage engines. Some of the commonly used MySQL storage engines are outlined in this lesson.

Note that InnoDB and NDBCLUSTER are the only two MySQL storage engines that are transactional.

Third-party engines have different sources and features, and are not supported by MySQL. For further information, documentation, installation guides, bug reporting, or any help or assistance with these engines, contact the developer of the engine directly. See the “Other Storage Engines” document at the following link for a brief discussion of third-party storage engines:

<http://dev.mysql.com/doc/mysql/en/storage-engines-other.html>

For more information about MySQL storage engines, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/storage-engines.html>.

# InnoDB Storage Engine

InnoDB, the default storage engine for MySQL, provides high reliability and high performance, as well as the following primary advantages:

- Transaction-safe (ACID compliant)
- MVCC (Multi-Versioning Concurrency Control)
  - InnoDB row-level locking
  - Oracle-style consistent non-locking reads
- Table data arranged to optimize primary key based queries
- Support for foreign-key referential integrity constraints
- Maximum performance on large data volumes
- Mixing of queries on tables with different storage engines
- Fast auto-recovery after a crash
- Buffer pool for caching data and indexes in memory

It is difficult for any other disk-related relational database engine to rival the efficiency of InnoDB. Here are some additional advantages of using InnoDB:

- **Transaction-safe:** ACID compliance is achieved with transaction commit, rollback, and crash-recovery capabilities to protect user data.
- **Foreign key support:** Includes cascaded deletes and updates
- **Recovery and backup:** Supports consistent and online logical backup
- **Mixing queries:** Within the same statement, you can mix InnoDB tables with tables from other MySQL storage engines. For example, you can use a `join` operation to combine data from InnoDB and MEMORY tables in a single query.
- **Full-text indexing:** Enables efficient searching for words or phrases within text columns

## InnoDB as Default Storage Engine

- Built in to the MySQL server
- Addresses many industry trends that demand an efficient storage engine:
  - Rising hard drive and memory capacity
  - Growing performance-to-price ratio requirement
  - Increased performance requiring reliability and crash recovery
  - Increasingly large, busy, robust, distributed, and important MySQL databases

The default storage engine for new tables is InnoDB. Choose an alternative storage engine when creating or altering tables with a clause such as `ENGINE=<Storage Engine>`.

## InnoDB Features

| Feature                       | Support            | Feature                            | Support |
|-------------------------------|--------------------|------------------------------------|---------|
| Storage limits                | 64 TB              | Index caches                       | Yes     |
| MVCC                          | Yes                | Data caches                        | Yes     |
| B-tree indexes                | Yes                | Adaptive hash indexes              | Yes     |
| Clustered indexes             | Yes                | Replication <sup>[c]</sup>         | Yes     |
| Compressed data               | Yes <sup>[a]</sup> | Update data dictionary             | Yes     |
| Encrypted data <sup>[b]</sup> | Yes                | Geospatial data type               | Yes     |
| Query cache                   | Yes <sup>[c]</sup> | Geospatial indexing                | No      |
| Transactions                  | Yes                | Full-text search indexes           | Yes     |
| Locking granularity           | Row                | Cluster database                   | No      |
| Foreign key                   | Yes                | Backup and recovery <sup>[c]</sup> | Yes     |
| File format management        | Yes                | Fast index creation                | Yes     |
| Multiple buffer pools         | Yes                | PERFORMANCE_SCHEMA                 | Yes     |
| Change buffering              | Yes                | Automatic crash recovery           | Yes     |

The table in the slide indicates whether InnoDB supports the listed features. Note the following qualifications:

- <sup>[a]</sup> Requires the InnoDB Barracuda file format
- <sup>[b]</sup> Implemented in the server (via encryption functions), rather than in the storage engine
- <sup>[c]</sup> Implemented in the server, rather than in the storage product. An exception is made when using MEB for backup, which is not at the server level.

# Displaying the Storage Engine Setting

- Use **SELECT** to confirm the session storage engine:

```
SELECT @@default_storage_engine;
```

- Use **SHOW** to confirm the storage engine, per a table:

```
SHOW CREATE TABLE City\G  
SHOW TABLE STATUS LIKE 'CountryLanguage'\G
```

- Use **INFORMATION\_SCHEMA** to confirm the storage engine, per a table:

```
SELECT TABLE_NAME, ENGINE FROM  
INFORMATION_SCHEMA.TABLES  
WHERE TABLE_NAME = 'City'  
AND TABLE_SCHEMA = 'world_innodb'\G
```

Confirm the setting by displaying the current value of the `storage_engine` variable:

```
mysql> SELECT @@default_storage_engine;  
+-----+  
| @@default_storage_engine |  
+-----+  
| InnoDB |  
+-----+
```

The SHOW examples displayed in the slide result in the following:

```
mysql> SHOW CREATE TABLE City\G  
***** 1. row *****  
Table: City  
Create Table: CREATE TABLE `City` (  
...  
) ENGINE=InnoDB AUTO_INCREMENT=4081 DEFAULT CHARSET=latin1
```

```
mysql> SHOW TABLE STATUS LIKE 'CountryLanguage'\G  
***** 1. row *****  
Name: CountryLanguage  
Engine: InnoDB  
...
```

# Setting the Storage Engine

- Set the server storage engine as part of the startup config file:

```
[mysqld]
default-storage-engine=<Storage Engine>
```

- Set for the current client session using the **SET** command:

```
SET @@storage_engine=<Storage Engine>;
```

- Specify using the **CREATE TABLE** statement:

```
CREATE TABLE t (i INT) ENGINE = <Storage Engine>;
```

If a table is created without using an **ENGINE** option to specify a storage engine explicitly, the MySQL server creates the table using the default engine, which is given by the value of the **storage\_engine** system variable.

You can overwrite the server default at the session level by using the **SET** command (as shown in the slide).

# Converting Existing Tables to InnoDB

Tables that are currently using other storage engines can be changed to InnoDB tables.

- Change storage engine using **ALTER TABLE**:

```
ALTER TABLE t ENGINE = InnoDB;
```

- Clone a table from another storage engine:
  - Create an empty InnoDB table with identical definitions.
  - Create the appropriate indexes.
  - Insert the rows:

```
INSERT INTO <innodb_table> SELECT * FROM  
<other_table>;
```

- Insert large tables in smaller pieces for greater control.

Using **ALTER TABLE** to change the engine is an expensive operation, because it copies all the data internally from one engine to another. Do not convert MySQL system tables in the `mysql` database (such as `user` or `host`) to InnoDB. This is an unsupported operation. The system tables use the MyISAM engine.

When you make an InnoDB table that is a clone of a table using another storage engine, you can also create the indexes after inserting the data.

To get better control over the insertion process, you might insert big tables in pieces:

```
INSERT INTO newtable SELECT * FROM oldtable  
WHERE yourkey > something AND yourkey <= somethingelse;
```

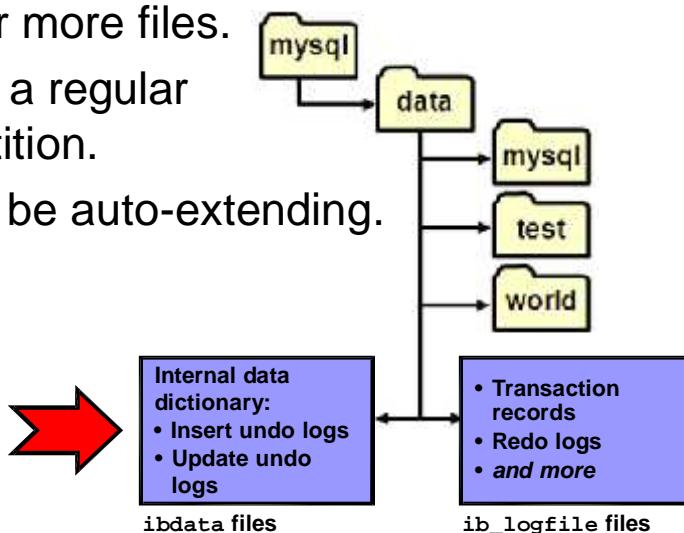
**Note:** After all records have been inserted, you can rename the tables.

During the conversion of big tables, increase the size of the InnoDB buffer pool to reduce disk I/O. You can also increase the sizes of the InnoDB log files.

It is best to run the MySQL server `mysqld` from the command prompt when you first start the server with InnoDB enabled, not from `mysqld_safe` or as a Windows service. For the steps involved, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/innodb-init.html>.

# InnoDB System Tablespace

- InnoDB metadata, undo log, and buffers are stored in a system “tablespace” by default.
- This is a single logical storage area, which can consist of one or more files.
- Each file can be a regular file or a raw partition.
- The last file can be auto-extending.



InnoDB operates using two primary disk-based resources:

- **Tablespace:** Stores table contents (data rows) and indexes in a single logical storage area
- **Log files:** Record transaction activity for rollback and recovery

InnoDB stores data, indexes, metadata, logs, and buffers in tablespaces. By default, data and indexes are stored in per-table tablespaces. InnoDB uses a shared tablespace to contain metadata, the undo log, the change buffer, and the doublewrite buffer.

The shared tablespace can occupy multiple files. You can configure the final file in the shared tablespace to be auto-extending, in which case InnoDB expands it automatically if the tablespace fills up.

The shared tablespace also contains a rollback segment by default. As transactions modify rows, undo log information is stored in the rollback segment. This information is used to roll back failed transactions. Move the rollback segment out of the shared tablespace file by setting the `innodb_undo_logs` option to a non-zero value, and configuring the value of `innodb_undo_tablespaces`.

**Note:** Data/tablespace buffer pool and log files are covered in detail later in this lesson.

# Data Tablespaces

- In addition to the system tablespace, InnoDB creates a further tablespace in the database directory—an `.ibd` file—for each InnoDB table.
- Per-table tablespaces provide the following benefits:
  - Table compression
  - Space reclamation (with `TRUNCATE`)
  - Dynamic row format
- (Optional) Store data in the shared tablespace:
  - Use the `skip_innodb_file_per_table` option.
  - Set at startup or during session.
- Use the shared tablespace for the following benefits:
  - Less file system overhead for statements that remove large amounts of data, such as `DROP TABLE` or `TRUNCATE TABLE`
  - Avoiding redundant temporary data files

Each new table that InnoDB creates sets up an `.ibd` file in the database directory to accompany the table's `.frm` file. The `.ibd` file acts as the table's own tablespace file, and InnoDB stores table contents and indexes.

You still need the shared tablespace because it contains the InnoDB data dictionary and the rollback segment.

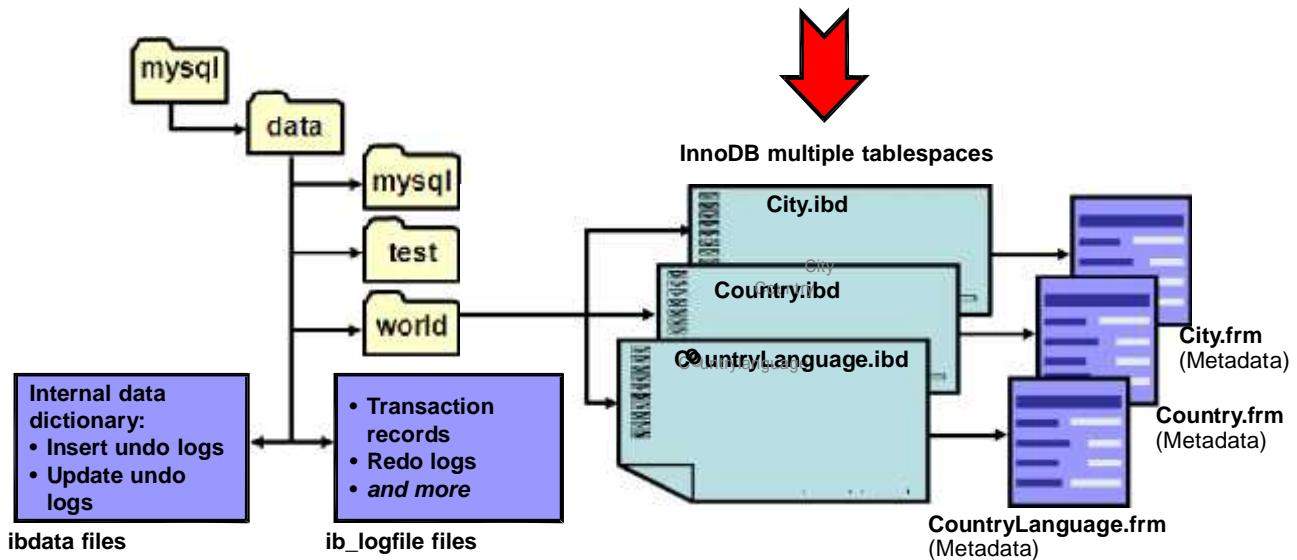
You can control this setting with the `innodb_file_per_table` option. You need the default setting to use some of the other features, such as table compression and fast truncation.

Rather than store data in per-table tablespaces, you can store data in the shared database by using the `skip_innodb_file_per_table` option or by setting the `innodb_file_per_table` option to OFF. Disabling the option does not affect accessibility of any InnoDB tables that have already been created. Those tables remain accessible.

You can mix the tablespace type among different tables in the same database. Changing the setting merely changes the default for new tables created, or for tables altered to set the engine to InnoDB (even tables that are already using InnoDB).

# Tablespace Directory Structure

Stored in the database directory



In addition to its tablespace files, the InnoDB storage engine manages a set of InnoDB-specific log files that contain information about ongoing transactions. As a client performs a transaction, the changes that it makes are held in the InnoDB log (ib\_logfile). The more recent log contents are cached in memory. Normally, the cached log information is written and flushed to log files on disk at transaction commit time, although that may also occur earlier.

# Shared Tablespace Configuration

- Increase the tablespace size by adding data files.
  - Use the `innodb_data_file_path` option in the `my.cnf` file.
    - The value must be a list of specifications:

```
[mysqld]
  innodb_data_file_path=datafile_spec1[,datafile_spec2]...
```

- Configuration example: Create a tablespace with a 50 MB (fixed) data file named `ibdata1` and an auto-extended 50 MB data file named `ibdata2`: [mysqld]

```
innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend
  – Files are placed in the data directory by default.
```
- Explicitly specify the file location, if desired.

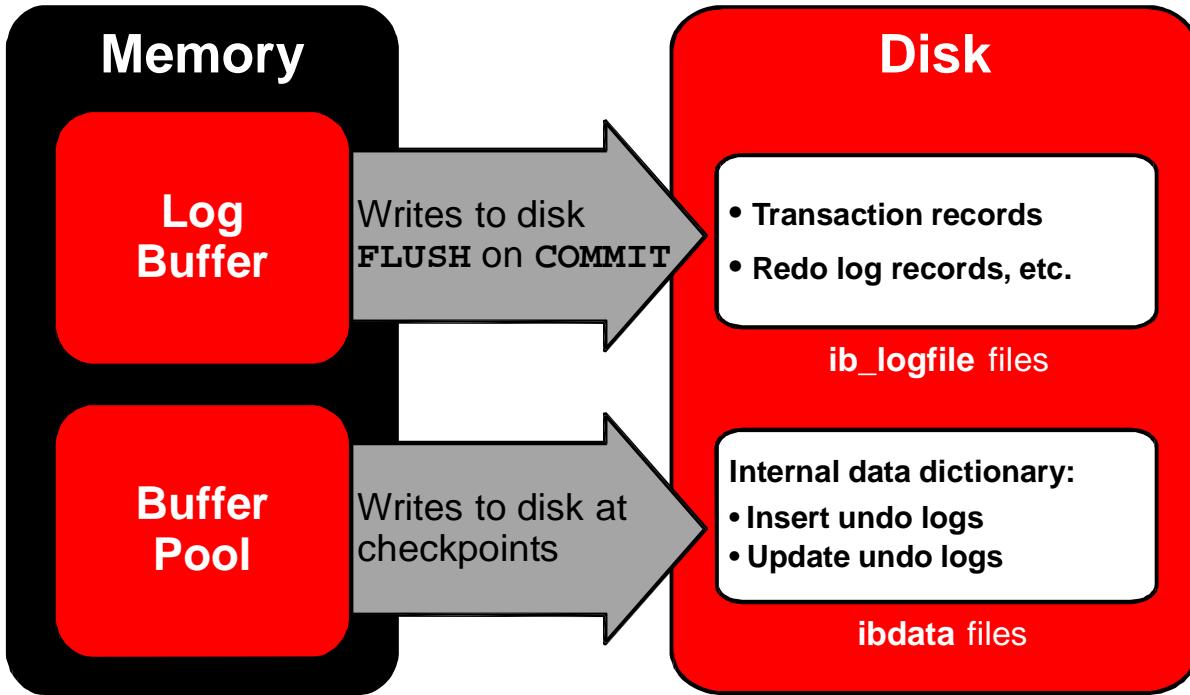
If more than one data file is named, separate them by semicolon (;) characters, as shown in the second slide example.

An easier way to increase the size of the InnoDB system tablespace is to configure it from the beginning to be auto-extending. Specify the `autoextend` attribute for the last data file in the tablespace definition. Then InnoDB increases the size of that file automatically in 64 MB increments when it runs out of space. You can change the increment size by setting the value of the `innodb_autoextend_increment` system variable, which is measured in MB. An added benefit of this approach is that you do not need to restart the server.

If your last data file was defined with the keyword `autoextend`, the procedure for reconfiguring the tablespace must take into account the size to which the last data file has grown. Obtain the size of the data file, round it down to the closest multiple of  $1024 \times 1024$  bytes (= 1 MB), and specify the rounded size explicitly in `innodb_data_file_path`. Then you can add another data file (using a file name that does not yet exist). Remember that only the last data file in the `innodb_data_file_path` can be specified as auto-extending.

For more information about InnoDB configuration parameters, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/innodb-parameters.html>.

## Log Files and Buffers: Diagram



As a client performs a transaction, the changes that it makes are held in the InnoDB log. The more recent log contents are cached in memory (log buffer). The cached log information is written and flushed to log files on disk at transaction commit time, although that may also occur earlier.

If a crash occurs while the tables are being modified, the log files are used for auto-recovery. When the MySQL server restarts, it reapplies the changes recorded in the logs to ensure that the tables reflect all committed transactions.

For information about changing the number or the size of your InnoDB log files, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/innodb-data-log-reconfiguration.html>.

InnoDB maintains its own buffer pool for caching frequently used data and indexes in main memory. InnoDB stores its tables and indexes in a tablespace. InnoDB tables can be very large, even on operating systems where file size is limited to 2 GB. You can enable multiple buffer pools to minimize contention.

This cache applies to so many types of information, and speeds up processing so much, that you should assign up to 80% of your database servers' physical memory to the InnoDB buffer pool.

If InnoDB configuration options are not specified, MySQL creates an auto-extending 10 MB data file named `ibdata1` and two 5 MB log files named `ib_logfile0` and `ib_logfile1` in the MySQL data directory. To get good performance, you should specify explicit InnoDB parameters.

You can choose a different number of InnoDB log files by using the `innodb_log_files_in_group` option. The default (and recommended) value is 2, resulting in the two files `ib_logfile0` and `ib_logfile1`.

To change the location of the `ib_logfile*` files, specify a directory path with the `innodb_log_group_home_dir` option. This is useful if you want to minimize the risk of hardware failure by storing log files on a different physical device to that which contains the data files.

# Configuring Buffer Pools for Performance

When the InnoDB buffer pool is large, many data requests can be quickly retrieved from memory rather than hard disk.

- MySQL uses multiple buffer pool instances to minimize contention between threads
- Each page that is stored in or read from the buffer pool is assigned to one of the buffer pools based on a hash value.
- Each buffer pool manages its own data.
- Configured with these options:
  - `innodb_buffer_pool_instances`
  - `innodb_buffer_pool_size`
- Preload buffer pools at startup to improve performance:
  - `innodb_buffer_pool_load_at_startup`
  - `innodb_buffer_pool_dump_at_shutdown`

MySQL uses multiple buffer pools as a performance enhancement for large buffer pools, typically in the multi-gigabyte range.

Each page in the buffer pool is assigned to one of the buffer pools randomly by using a hashing function. Each buffer pool manages its own free lists, flush lists, LRU's, and all other data structures connected to a buffer pool, and is protected by its own buffer pool mutex.

By default, MySQL configures eight buffer pool instances (except on 32-bit Windows systems). To change this, set the `innodb_buffer_pool_instances` configuration option to a value from 1 (the minimum) to 64 (the maximum). This option takes effect only when you set the `innodb_buffer_pool_size` to a size of 1 GB or more. The total size that you specify is divided among all the buffer pools. You should specify a combination of `innodb_buffer_pool_instances` and `innodb_buffer_pool_size` so that each buffer pool instance is at least 1 GB.

To preload the buffer pool at server restart, enable the options `innodb_buffer_pool_dump_at_shutdown` and `innodb_buffer_pool_load_at_startup`.

# NoSQL and the memcached API

An embedded **memcached** daemon:

- Operates in the same process space
- Uses the memcached API
  - Simple, well-known, efficient
  - Bypasses SQL layer, avoiding the overhead of:
    - Parsing
    - Optimizing execution plans
    - Dealing with strongly typed data
- Uses InnoDB storage
  - Persisted to disk automatically
  - Benefits of buffers and crash recovery: protects against crashes, outages, and corruption

InnoDB supports a plugin with an integrated memcached daemon, which operates in the same process space for efficient data communication with low overhead. By using the simpler memcached API (both text-based and binary protocols) instead of using the SQL layer for some operations, your applications can bypass the parsing and optimizing stages required when submitting SQL statements, and avoid the overhead of checking strongly-typed data. This type of application is called *NoSQL* (Not Only SQL).

In addition to the classic memcached environment, which loses the key/value store when memcached is restarted, the InnoDB integration means you can configure the memcached store to persist to InnoDB. Because the store is backed by InnoDB, you benefit from other features of InnoDB such as buffered data access and crash recovery. This means you get the speed and simplicity of using a memory-stored memcached store, backed by a robust and persistent InnoDB table.

You can still access the underlying table through SQL for reporting, analysis, ad hoc queries, bulk loading, set operations (such as union and intersection), and other operations (such as summarizing and aggregation) that are well suited to the expressiveness and flexibility of SQL.

Because memcached consumes relatively little CPU, and its memory footprint is easy to control, it can run comfortably alongside a MySQL instance on the same system.

# Configuring memcached

- Configure memcached tables:

```
SOURCE <MySQL_HOME>/scripts/innodb_memcached_config.sql
```

- Creates the `innodb_memcache` database, which contains the following tables:
  - `cache_policies`: Holds the backing store policy for each memcached operation; local RAM cache, InnoDB, or both
  - `Containers`: Holds the configuration of each InnoDB table used as a memcached backing store
  - `config_options`: Holds values for the `separator` and `table_map_delimiter` configuration options
- Install the memcached plugin:

```
INSTALL PLUGIN daemon_memcached SONAME "libmemcached.so";
```

- Configure memcached variables:

- `daemon_memcached_option`: Space-separated options passed to the memcached daemon at startup

The `innodb_memcached_config.sql` script sets up the memcached configuration database and its tables.

The default InnoDB backing store is the `demo_test` table in the `test` database. You can change this by modifying the `containers` table in the `innodb_memcache` database.

The plugin enables a number of configurable variables. To view the memcached configuration variables, issue the following command:

```
SHOW VARIABLES LIKE 'daemon_memcached%';
```

For example, the following options configure the behavior of the memcached plugin:

- `daemon_memcached_enable_binlog`: Enable binary logging for memcached operations, which enables replication of memcached API operations.
- `daemon_memcached_r_batch_size`: Control the read batch size, for performance considerations.
- `daemon_memcached_w_batch_size`: Control the write batch size. The default value is 1, so each memcached write operation is committed immediately.

## Key/Value Data

The memcached protocol supports operations such as **add**, **delete**, **set**, **get**, **incr**, and **decr**.

- Each operation works with key/value pairs in the cache.
  - Keys and values are passed as strings.
  - Keys cannot contain whitespace characters such as spaces, carriage returns, newlines, tabs, and so on.
  - A key is similar to a primary key in a table, and a value is similar to a second column in the same table.
- Cache is a flat namespace.
  - Ensure that each key is unique. For example, if you are representing key/value pairs from different tables, prepend the table name to the key.
- A value can be a simple string (or number stored as a string) or serialized complex data.

The main benefit of using memcached is its efficiency when reading or writing key/value data. Conceptually, this is similar to a two-column table where the key column is a primary key (and therefore unique). Both key and value are stored as strings, and due to the nature of the memcached protocol, key values cannot contain spaces or newlines.

The direct, low-level database access path used by the memcached plugin is much more efficient for key/value lookups than equivalent SQL queries.

The value is stored as a string, and can represent a simple value such as a number or a string, or a complex value that has been serialized (that is, put in a textual form that can reconstruct the original pointers and nesting). You must perform this serialization outside of memcached at the API interface (for example, by implementing the Java `Serializable` interface in your code) or by formatting the complex data as JSON or XML.

Similarly, simple numeric values must be converted from string to number in SQL statements that access the `memcached` table (for example, by using the `CAST( )` function).

# Using NoSQL with MySQL Databases

- Use InnoDB tables as a container (backing store) for your memcached operations.
- Keys and values are strings.
  - Keys have a maximum size of 250 bytes. For longer keys:
    - Hash the key value to a value less than 250 bytes
    - Reconfigure the allowed size for memcached keys
- Use multiple InnoDB tables as memcached containers by specifying different classes of memcached data.
  - Configure this in the `containers` table.
  - For example, issue a memcached request in the form `get @@newcontainer` to redirect subsequent requests to the container named “newcontainer”.

To get the maximum benefit from memcached, use it as a fast key/value access method, with an InnoDB backing store. The memcached protocol does not support easy aggregation, joining data, or other features that the SQL protocol supports. Using InnoDB tables as containers for memcached key/value pairs allows you to execute powerful SQL statements performing grouping, aggregation, and joining of data created and modified with the memcached protocol.

The `innodb_memcache.containers` table allows you to configure one or more InnoDB tables to use as memcached backing stores. Specify the container table’s name, schema, and columns for key, value, flags, and other settings. Each container has a memcached name, which you can use to identify the container for memcached operations. Operations target the container named “default” until requested to change container.

The `/usr/share/mysql/innodb_memcached_config.sql` file creates and populates the `innodb_memcache.containers` table, and is a well-commented example of configuring a container.

# Referential Integrity

*Referential integrity* means that relationships between tables follow specified rules.

- Relationships are enforced by *foreign key constraints*.
  - A column in the child table (the *foreign key*) relates to a column in the parent table.
  - The parent column must be contained within a primary key or unique index.
- Relationships determine the results of SQL statements:
  - An **INSERT** into a child table (without a foreign key to the parent table) does not complete.
  - Using foreign keys on joined columns improves performance.
  - When you **UPDATE** or **DELETE** data, related data in the linked table can be automatically updated or deleted with the **CASCADE** option.

Referential integrity means that relationships between tables are consistent. MySQL enforces referential integrity by using foreign key constraints. When one table (the *child* table) has a foreign key to another table (the *parent* table), MySQL prevents you from adding a record to the child table if there is no corresponding record in the parent table. It also facilitates cascading updates and deletes to ensure that changes made to the child table are reflected in the parent table.

## Using Foreign Keys

- Specify a key for every table in the relationship by using the most frequently queried column or columns, or an auto-increment value if there is no obvious primary key.
  - A primary key is not mandatory, but there must be an index in the parent table which lists the referenced columns first, in the same order.
- For fast join performance, define foreign keys on the join columns, and declare those columns with the same data type in each table.

## Referential Actions

If you attempt to insert or update data in a child column, and that value does not exist in the parent column, the operation fails. If you attempt to change a value in the parent column such that a dependant child column would be affected by that change, the result depends on the *referential action*.

Specify the referential action within ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause.

- By default, the insert or delete operation fails and you receive an error. This is the same as specifying RESTRICT or NO ACTION.
- If you use CASCADE, foreign keys propagate deletes or updates to all affected tables.
- If you use SET NULL, values in the foreign key column(s) of the child table are set to NULL instead of the new parent key value.

The following example relates parent and child tables through a single-column primary key. The foreign key contains an ON DELETE CASCADE subclause to ensure that if you delete a record in the parent table, MySQL deletes any corresponding records in the child table:

```
CREATE TABLE p1 (
    id
    INT NOT NULL,
    PRIMARY KEY (id)
) ENGINE=INNODB;

CREATE TABLE c1 (
    id INT,
    parent_id INT,
    INDEX par_ind (parent_id),
    FOREIGN KEY (parent_id)
        REFERENCES p1(id) ON
        DELETE CASCADE
) ENGINE=INNODB;
```

# Multi-Versioning

- Keeps information about old versions of changed rows
  - Supports transactional features such as concurrency and rollback
- Stores information in the tablespace in a data structure called a *rollback segment*
- InnoDB adds three fields to each row stored in a database:
  - **DB\_TRX\_ID**: Transaction identifier
  - **DB\_ROLL\_PTR**: Roll pointer
  - **DB\_ROW\_ID**: Row ID
- Physically removes a row (and its indexes) only when it discards the update undo log record written for the deletion:
  - This is a fast purge operation.
  - The purge thread might need more resources because it can get slow and take disk space.

A *rollback segment* is an InnoDB storage area that contains the undo log. InnoDB can respond to queries for multiple versions of the same row when those queries are part of transactions that started at different times. It uses part of the undo log, the *update undo buffer*, to build earlier versions of database rows.

Three hidden fields exist on every row of user data managed by InnoDB:

- **DB\_TRX\_ID**: Six-byte field that indicates the *transaction identifier* for the last transaction that inserted or updated the row. Also, a deletion is treated internally as an update where a special bit in the row is set to mark it as deleted.
- **DB\_ROLL\_PTR**: Seven-byte field called the *roll pointer*. It points to an undo log record written to the rollback segment. If the row was updated, the undo log record contains the information necessary to rebuild the content of the row before it was updated.
- **DB\_ROW\_ID**: Six-byte field that contains a *row ID* that automatically increments as new rows are inserted. If InnoDB generates a clustered index automatically, the index contains row ID values. Otherwise, the **DB\_ROW\_ID** column does not appear in any index.

If you insert and delete rows in small batches at the same rate in the table, the purge thread can lag behind and the table can grow bigger and bigger because of all the “dead” rows, making everything disk-bound and very slow. In such a case, throttle new row operations and allocate more resources to the purge thread by tuning the `innodb_max_purge_lag` system variable.

# Locking

InnoDB locking does not need to set locks to achieve consistent reads.

- Uses row-level locking for DML statements
- Never escalates locks
- Wait-for graph detection for deadlocks

InnoDB has the following general locking properties:

- InnoDB does not need to set locks to achieve consistent reads because it uses multi-versioning to make them unnecessary. Transactions that modify rows see their own versions of those rows, and the undo logs allow other transactions to see the original rows. To force `SELECT` statements to lock data, you can add locking modifiers to the statements.
- When locks are necessary, InnoDB uses row-level locking. In conjunction with multi-versioning, this results in good query concurrency because a given table can be read and modified by different clients at the same time.
- InnoDB may acquire row locks as it discovers them to be necessary. It never escalates a lock by converting it to a page lock or table lock. This keeps lock contention to a minimum and improves concurrency (although it does use table-level locking for DDL operations).
- Deadlock is possible because InnoDB does not acquire locks during a transaction until they are needed. InnoDB can detect a deadlock and roll back one transaction to resolve the deadlock.
- Failed transactions eventually begin to time out, and InnoDB rolls them back as they do.

For more information about deadlocks, see the glossary of MySQL terms at [http://dev.mysql.com/doc/mysql/en/glossary.html#glos\\_deadlock](http://dev.mysql.com/doc/mysql/en/glossary.html#glos_deadlock).

## Next-Key Locking

- InnoDB uses an algorithm called *next-key locking*, which:
  - Uses row-level locking
  - Searches/scans for table index and sets shared or exclusive locks on index records
- Next-key locks affect the “gap” before the index record.
  - A new index record cannot be inserted right before a locked record.
- Locking of gaps prevents the “phantom problem.”
  - In the following example, the first transaction locks any values greater than 10, even if they do not exist:

**Transaction 1:**

```
SELECT c FROM t WHERE c > 10 FOR UPDATE;
```

**Transaction 2:**

```
INSERT INTO t(c) VALUES (50);
```

InnoDB uses an algorithm called *next-key locking* with row-level locking. The locking is performed in such a way that when an index of a table is searched or scanned, it sets shared or exclusive locks on the index records encountered. Thus, the row-level locks are actually index record locks. The next-key locks that InnoDB sets on index records also affect the “gap” before that index record. If a user has a shared or exclusive lock on a record in an index, another user cannot insert a new index record immediately before the locked record (the gap) in the index order.

This next-key locking of gaps is done to prevent the so-called *phantom problem*. The phantom problem occurs within a transaction when the same query produces different sets of rows at different times. For example, if a `SELECT` is executed twice but returns a row the second time that was not returned the first time, the row is a “phantom” row.

# Consistent Non-Locking Reads

| Time | Session 1   | Session 2                        |
|------|---|----------------------------------|
|      | s1> START TRANSACTION;  | s2> START TRANSACTION;           |
|      | s1> SELECT * FROM t;<br>Empty Set                                   |                                  |
|      |   | s2> INSERT INTO t VALUES (1, 2); |
| V    | s1> SELECT * FROM t;<br>Empty Set                                   |                                  |
|      |   | s2> COMMIT;                      |
|      | s1> SELECT * FROM t;<br>Empty Set                                   |                                  |
|      | s1> COMMIT;   |                                  |
| V    | s1> SELECT * FROM t;<br>-----<br>  1   2  <br>-----<br>1 row in set |                                  |

A consistent read means that InnoDB uses multi-versioning to present your query with a snapshot of the database at a point in time. Your query sees the changes made by transactions that committed before that point of time, and does not see changes made by later or uncommitted transactions.

In the slide example, Session 1 sees the row inserted by Session 2 only when 2 has committed the insert and 1 has committed as well, so that the timepoint is advanced past the commit of 2.

If you want to see the “freshest” state of the database, use either the READ COMMITTED isolation level or a locking read.

# Reduce Deadlocks

- Use transactions rather than **LOCK TABLE** statements.
  - Keep transactions small and commit them often.
  - Use the same order of operations when different transactions update multiple tables.
  - Create indexes on columns.
- Re-issue a transaction if it fails due to deadlock.
  - Deadlocks are not dangerous. Just try again.
- Access your tables and rows in a fixed order.
- Add well-chosen indexes to your tables.
- Do not set the isolation level to avoid deadlocks.
  - This has no effect because it is not related to read operations.
- Monitor how often deadlocks occur.
  - Use the **SHOW ENGINE INNODB STATUS** command.

To reduce the possibility of deadlocks, use transactions rather than **LOCK TABLE** statements:

- Keep transactions that insert or update data small enough (a small number of rows) that they do not stay open for long periods of time. Commit your transactions often.
- When different transactions update multiple tables or large ranges of rows, always try to reference those tables or rows in the same sequence.
- Create indexes on the columns used in the **WHERE** clause.

The isolation level changes the behavior of read operations, while deadlocks occur because of write operations.

If a deadlock does occur, InnoDB detects the condition and rolls back one of the transactions (the victim). Therefore, even if your application logic is perfectly correct, you must still handle the case where a transaction must be retried. To monitor how frequently deadlocks occur (as well as many other InnoDB stats), use the **SHOW ENGINE INNODB STATUS** command:

```
mysql> SHOW ENGINE INNODB STATUS\G
=====
110222 16:54:12 INNODB MONITOR OUTPUT
=====
Per second averages calculated from the last 5 seconds
...
```

# Foreign Key Locking

InnoDB supports foreign key **CONSTRAINT** settings, which:

- Require checking the constraint condition
- Are used for **INSERT**, **UPDATE**, and **DELETE**
- Set shared record-level locks on records to check
- Also set locks on cases where a constraint fails

Add foreign key constraints when creating a table:

```
CREATE TABLE City (
    ...
    CountryCode char(3) NOT NULL DEFAULT '',
    ...
    KEY CountryCode (CountryCode),
    CONSTRAINT city_ibfk_1 FOREIGN KEY
        (CountryCode) REFERENCES country (Code)
    ...
)
```

A constraint is simply a restriction placed on one or more column values of a table to actively enforce integrity rules. Constraints are implemented using indexes.

If a foreign key constraint is defined on a table, a shared record level lock is placed on any record that is used in an insert, update, or delete operation that references foreign key constraints. InnoDB also sets these locks in the case where the constraint fails.

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

The constraint example in the slide shows that the `CountryCode` column of the `City` table is related to the `Code` column of the `Country` table. Any changes to either are constrained by this relationship.

## **MEMORY Storage Engine**

## **MyISAM Storage Engine**

The MyISAM storage engine stores each table on disk in three files (**.frm**, **.MYD**, and **.MYI**) and has the following features:

- Support for **FULLTEXT** searching and spatial data types
- Flexible **AUTO\_INCREMENT**
- Compressed, read-only tables, which save space
- Table-level locking to manage contention between queries
- Portable storage format
- Ability to specify the number of rows for a table
- Ability to control the updating of non-unique indexes for loading data into an empty table

MyISAM was the default MySQL storage engine prior to the MySQL server version 5.5.5. The current default is the InnoDB storage engine. The `mysql` database contains tables in the MyISAM format.

Each MyISAM table is represented by three files:

- **Format file:** Stores the definition of the table structure (`mytable.frm`)
- **Data file:** Stores the contents of table rows (`mytable.MYD`)
- **Index file:** Stores any indexes on the table (`mytable.MYI`)

Additional features:

- MyISAM tables take up very little space.
- The table storage format is portable, so table files can be copied directly to another host and used by a server on that host.
- When loading data into an empty table, updating of non-unique indexes can be disabled and then re-enabled after loading the data.
- MyISAM supports geometric spatial extensions.
- MyISAM can improve performance by limiting table size to a certain number of rows.

# MEMORY Storage Engine

The MEMORY storage engine creates tables with contents that are stored in memory, represented on disk by a `.frm` file.

It has the following features:

- Table data and indexes that are stored in memory
- Very fast performance due to in-memory storage
- Fixed-length row storage format
- Table contents that do not survive restart
- Maximum size option `--max-heap-table-size`
- Table-level locking

MEMORY tables:

- Cannot contain `TEXT` or `BLOB` columns
- Can use different character sets for different columns

- Each table is represented on disk by an `.frm` format file in the database directory.
- Contents do not survive a restart of the server (the structure survives, but the table contains zero rows).
- MySQL manages query contention by using table-level locking. Deadlock cannot occur.

HEAP may still be seen in older SQL code, and the MySQL server still recognizes HEAP for backward compatibility.

MEMORY performance is constrained by contention resulting from single-thread execution and table-lock overhead when processing updates. This limits scalability when the load increases, particularly for statement mixes that include writes. Also, MEMORY does not preserve table contents across server restarts.

MEMORY is a valid and useful storage engine and should be considered with almost any application design to improve performance and to meet the needs of specific business rules. For more information about MEMORY storage engine features, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/memory-storage-engine.html>.

## ARCHIVE Storage Engine

The ARCHIVE storage engine is used for storing large volumes of data in a compressed format, allowing for a very small footprint. It has these primary characteristics:

- Represented by **.frm** file
- Data file: **.ARZ**
- Does not support indexes
- Supports **INSERT** and **SELECT**, but not **DELETE**, **REPLACE**, or **UPDATE**
- Supports **ORDER BY** operations and **BLOB** columns
- Accepts all but spatial data types
- Uses row-level locking
- Supports **AUTO\_INCREMENT** columns

The BLACKHOLE storage engine acts as a “black hole” that accepts data but throws it away and does not store it. This storage engine has these primary characteristics:

- Represented by **.frm** file
- Used for replication
- Supports all kinds of indexes
- Retrievals always return an empty result.
- Verification of dump file syntax
- Measurement of the overhead from binary logging
- “No-op” storage engine that can be used for finding performance bottlenecks not related to the storage engine
- Transaction-aware

Committed transactions are written to the binary log, and rolled-back transactions are not.

## **Chapter 5**

### **Partitioning**

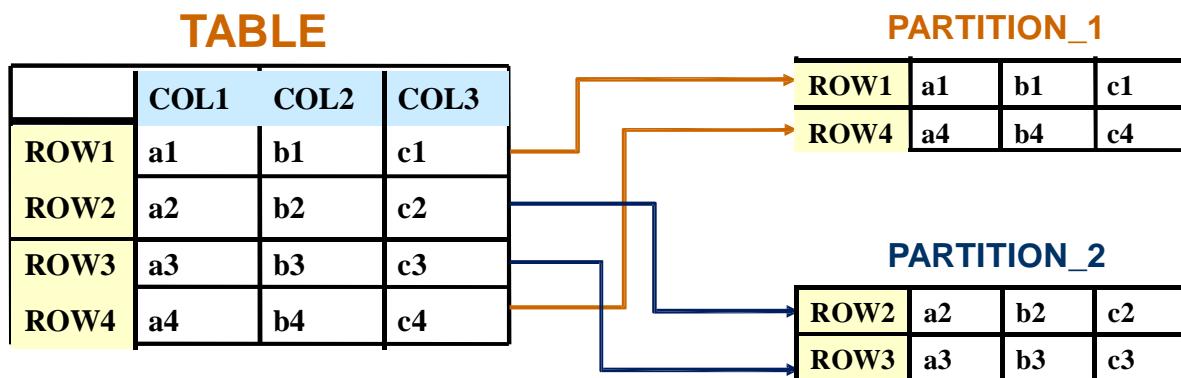
# **Objectives**

After completing this lesson, you should be able to:

- Define partitioning and its particular use in MySQL
- Determine server partitioning support
- List the reasons for using partitioning
- Explain the types of partitioning
- Create partitioned tables
- Describe subpartitioning
- Obtain partitioning metadata
- Use partitioning to improve performance
- Explain storage engine implementation of partitioning
- Explain how to lock a partitioned table
- Describe partitioning limitations

# Partitioning

- A partition is the division of a database, or its constituting elements, into distinct independent parts.
  - A method of pre-organizing table storage
- MySQL supports horizontal partitioning.
  - Assigns specific table rows into subsets of rows

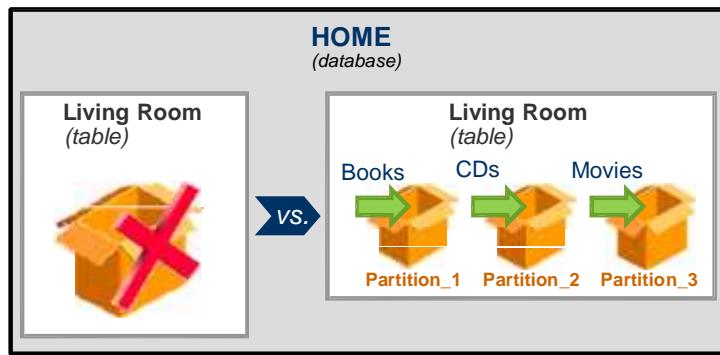


This lesson focuses on the MySQL partitioning technique known as “horizontal partitioning,” as illustrated in the slide.

Physical partitioning is used when large tables are taxing the available disk space, and additional space is needed to store the table data. Table files can be placed in multiple locations, rather than forcing the data onto an overloaded disk. This technique is storage engine-dependent.

# Partitioning Analogy

- The “moving” analogy:
  - Do you want to put everything from the living room into one giant box?
  - How about putting items in several smaller boxes, by type?
  - You have now partitioned your belongings!
- The “Home” database example:



The “moving” analogy is being used to portray partitioning. Suppose that you are moving out of your home. You could put all the things from your living room in one big box. However, that would not be very practical. It would be very difficult for you to find individual items. And it would be a very large and heavy box! It would be much better to use several boxes to hold all your items, and to categorize those boxes according to type. For example, you could have a box labeled “Books” for all your books, a box for “CDs,” a box for “Movies,” and so on. Now you have *partitioned* your belongings into specific boxes. If this were a table structure, it would look something like the example in the slide.

# MySQL-Specific Partitioning

- Distribution of partitions is across physical storage.
  - According to user-specified rules that are set as needed
  - Each partition is stored as its own unit.
- Division of data
  - Data is divided into subsets based on a partitioning function.
  - The partitioning type and expression are part of the table definition.
  - The expression can be an integer or a function that returns an integer value.
  - This value determines in which partition to store each record, per definition.

MySQL partitioning allows you to distribute portions of individual tables across your physical storage, according to rules that you can set largely as needed. Each partition is stored as a separate unit, much like a table. The way that MySQL accomplishes this is as follows:

- The division of data is accomplished with a partitioning function, which in MySQL can be a simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function.
- The function is selected according to the partitioning type specified by the user, and takes the value of a user-supplied expression.
- The expression can be either an integer column value, or a function acting on one or more column values and returning an integer.
  - The expression must return `NULL` or an integer value.
  - With the use of `RANGE/LIST COLUMNS`, the expression can be one or more columns of the following data types: `INTS`, `DATE`, `DATETIME`, `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY`.
- This value determines which partition each record should be stored in, according to the partition definition(s).

For more information about MySQL partitioning, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/partitioning.html>.

# Partitioning Types

| Type  | Description  | Variant | Description  |
|-------|--|---------|--|
| RANGE | Assigns rows to partitions based on column values falling within given ranges  | COLUMNS | Enables the use of one or more columns in RANGE and LIST partitioning keys |
| LIST  | Assigns rows to partitions based on columns matching one of a set of discrete values   |         |  |
| HASH  | Partitions selected based on the value returned by a user-defined expression, operating on column values in rows to be inserted into the table | LINEAR  | Uses a linear powers-of-two algorithm for HASH and KEY partitions          |
| KEY   | Similar to HASH, except that only one or more columns to be evaluated are supplied   |         |  |

MySQL supports several types of partitioning:

- **RANGE**: Ranges should be contiguous but not overlapping, and are defined using the `VALUES LESS THAN` operator.
- **LIST**: As in partitioning by RANGE, each partition must be explicitly defined.
- **HASH**: Operating on column values in rows to be inserted into the table
- **KEY**: Similar to HASH, except that only one or more columns to be evaluated are supplied, and the MySQL server provides a hashing function. It works on all allowed column types.
- **COLUMNS**: Variants on RANGE and LIST partitioning. COLUMNS partitioning enables the use of one or more columns in partitioning keys. All of these columns are taken into account both for the purpose of placing rows in partitions and for the determination of which partitions are to be checked for matching rows in partition pruning.
  - **RANGE COLUMNS** and **LIST COLUMNS** partitioning support the use of non-integer columns (and other data types listed earlier) for defining value ranges or list members.
- **LINEAR**: MySQL also supports linear hashing, which differs from regular hashing in that linear hashing uses a linear powers-of-two algorithm whereas regular hashing employs the modulus of the hashing function's value.

# Partitioning Support

- Determine server partitioning support status:

```
mysql> SHOW PLUGINS\G
...
***** row *****
Name: partition
Status: ACTIVE
Type: STORAGE ENGINE
Library: NULL
License: PROPRIETARY
...
...
```

- Disable partitioning support:

```
shell> mysqld --skip-partition
```

- The `partition` plugin now has the value `DISABLED`.

If the MySQL binary is built with partitioning support, you do not need to take any further action to enable the binary (for example, no special entries are required in your `my.cnf` file).

If you do not see the `partition` plugin with the Status value `ACTIVE` listed in the output of `SHOW PLUGINS`, then your version of MySQL does not support partitioning.

# Improving Performance with Partitioning

- Address problems specific to large tables:
  - Break data into smaller groups for more efficient queries.
  - Adhere to file-system file size limits.
  - Speed up runtimes for maintenance and deleting rows.
- Use smaller indexes to permit the “hottest” partition to have the whole index in memory.
- Drop a partition to make removing data easier.
- Segregate tables according to date and time.

When you split data into logically grouped tables, a query has fewer records to search. This greatly improves the speed. Partitioning addresses some potential problems specific to large tables, such as slow runtimes for maintenance operations (for example, `OPTIMIZE` and `REPAIR`) and slow runtime when deleting unnecessary rows.

It is natural to want to partition tables according to date and time values. For example, you might want to segregate a table of store orders by year, quarter, or month. With MySQL, you can use partitioning schemes that are based on dates and times, making your queries more efficient.

# Improving Performance with Partitioning: Pruning

- Prune data to confine the query scope to precise ranges:
  - MySQL optimizer filters out unneeded partitions for a query.
  - Full table scans using the **WHERE** clause prune away partitions as non-matching.

```
SELECT title FROM book
WHERE category = 'medicine'
AND published > 1990 AND published < 2007;
```

- Perform explicit partition selection.

```
SELECT title FROM book PARTITION(p0)
WHERE category = 'medicine'
AND published > 1990;
```

- Also use explicit partition selection with **INSERT**, **REPLACE**, **UPDATE**, **DELETE**, and **LOAD**.

“Pruning” happens when the MySQL optimizer deduces that the rows it needs to access to execute a particular query reside in a limited number of partitions. It searches only the partitions that fit the criteria of the query.

As the first example in the slide shows, suppose that you have a bookstore and you have a database containing the entire inventory of books in your store. This database has tables for each category of books. A customer comes in to your store and asks for a list of all the books in the category of medicine, published between the years of 1990 and 2007. Because you took advantage of partitioning and have your medicine table partitioned by year, you can run a query that includes only those years. The optimizer “prunes” the search to scan only the 1990–2007 partitions, saving a huge amount of time and effort. (Note that MySQL does not transform the actual table data in any way.)

You can also perform explicit partition selection in statements that reference table data. For example, if you know that `p0` includes all books published before 2007, you can explicitly state the partition to allow the optimizer to ignore other partitions. You can select multiple partitions and subpartitions by providing a list—for example `PARTITION(p0, p2sp1)`.

Explicit partition selection also works with **INSERT**, **REPLACE**, **UPDATE**, **DELETE**, and **LOAD** statements, although if you are adding or changing data in a partition, you must ensure that the new data meets the partitioning criteria for that partition.

## Performance Challenges with Partitioning

All partitions must be opened when opening a partitioned table.

- This is a minor problem because opened tables are cached in the table cache.

Pruning presents specific restrictions:

- Pruning is less efficient on indexed columns because it avoids only one index lookup per partition.
- The effect of pruning is much greater for non-indexed columns.
  - A query with a non-indexed column in its WHERE clause might require a full table scan.
  - A partition scan takes a fraction of the time that a table scan takes.

## Basic Partition Syntax

Create partitioned tables by using the **PARTITION BY** clause in the **CREATE TABLE** statement. **PARTITION BY** comes after all parts of **CREATE TABLE**:

```
CREATE TABLE <table_name> (<table_column_options>)
ENGINE=<engine_name>
PARTITION BY <type> (<partition_expression>);
```

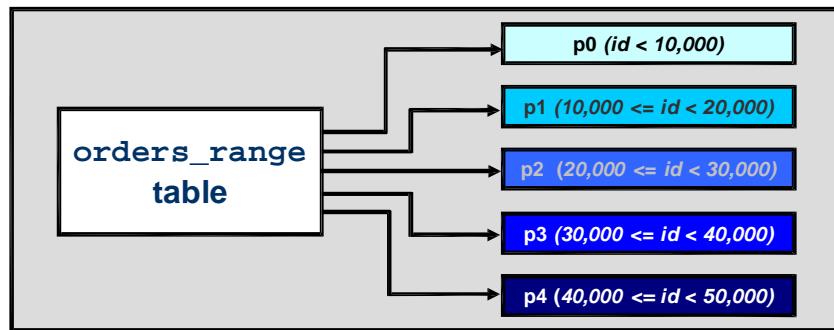
The partitioning type is followed by a partitioning expression in parentheses. Depending on the type of partitioning you use and your requirements, this expression can be the name of a table column, a simple mathematical expression involving one or more columns, or a comma-separated list of column names.

**PARTITION BY** types as included in their corresponding syntax:

- **PARTITION BY RANGE ...**
- **PARTITION BY RANGE COLUMNS ...**
- **PARTITION BY LIST ...**
- **PARTITION BY LIST COLUMNS ...**
- **PARTITION BY HASH ...**
- **PARTITION BY LINEAR HASH ...**
- **PARTITION BY KEY ...**
- **PARTITION BY LINEAR KEY ...**

## RANGE Partitioning

- Use **PARTITION BY RANGE** to partition rows that contain specified values within a set of specified ranges.
- In this “store orders” example, search ranges are specified for the data related to the orders:
  - A partition for orders with ID numbers less than 10,000
  - A partition for orders with ID numbers between 10,000 and 19,999
  - And so on, through 50,000



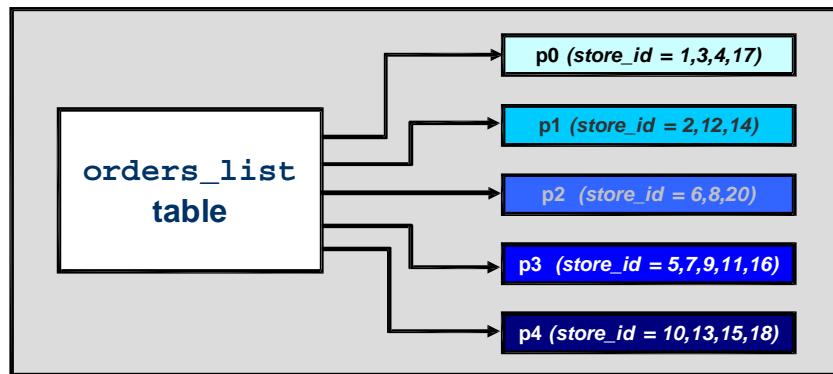
Rows with column or expression values falling within a specified range are assigned to a given partition. For example, suppose you have a table that stores data related to its orders, and you want to specify that orders with `id` numbers less than 10,000 be stored in one partition, orders with `id` numbers between 10,000 and 19,999 in another, orders with `id` numbers between 20,000 and 29,999 in another, and so on. In other words, you partition the table based on contiguous ranges of values. RANGE partitioning is a good fit for such a partitioning scheme. For this case, you would use the following syntax to create a new table that is set up for range partitioning:

```
mysql> CREATE TABLE orders_range (
->   id INT AUTO_INCREMENT PRIMARY KEY,
->   customer_surname VARCHAR(30),
->   store_id INT, salesperson_id INT,
->   order_date DATE, note VARCHAR(500)
-> ) ENGINE = InnoDB
-> PARTITION BY RANGE(id) (
->   PARTITION p0 VALUES LESS THAN(10000),
->   PARTITION p1 VALUES LESS THAN(20000),
->   PARTITION p2 VALUES LESS THAN(30000),
->   PARTITION p3 VALUES LESS THAN(40000),
->   PARTITION p4 VALUES LESS THAN(50000)
-> );
```

A RANGE partitioned table accepts only rows that match a partition.

## LIST Partitioning

- Use **PARITION BY LIST** to partitions rows that contain specified values.
- In this “store orders” example, partitions are specified for orders by store IDs, per district:



You specify lists of values for each partition so that rows with matching column values are stored in the corresponding partition. For instance, assume that each of your orders is placed at one of several different stores organized into five districts, and you would like to store orders placed at each district in the same partition. Using this type of partitioning allows you to arbitrarily separate the stores. Suppose that you have 19 stores in these five districts, organized as shown in the slide.

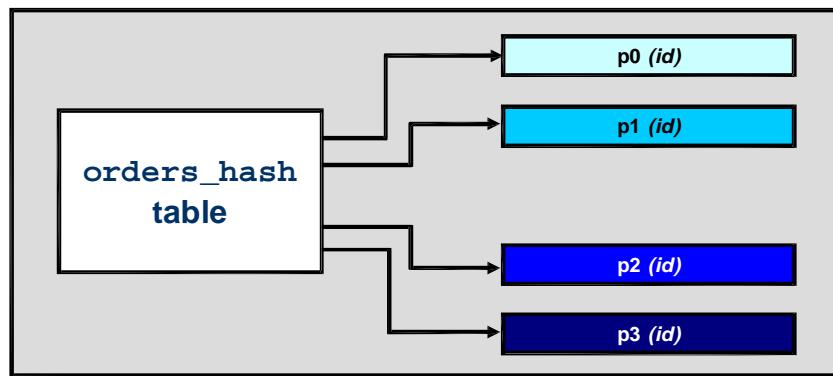
You can create an `orders_list` table that assigns orders coming from stores in the same district to the same partition (as the diagram in the slide shows) as follows:

```
mysql> CREATE TABLE orders_list (
    -> id INT AUTO_INCREMENT, customer_surname VARCHAR(30), store_id INT,
    -> salesperson_id INT, order_date DATE, note VARCHAR(500),
    -> INDEX idx (id)) ENGINE = InnoDB
    -> PARTITION BY LIST(store_id) (
    -> PARTITION p0 VALUES IN (1, 3, 4, 17),
    -> PARTITION p1 VALUES IN (2, 12, 14),
    -> PARTITION p2 VALUES IN (6, 8, 20),
    -> PARTITION p3 VALUES IN (5, 7, 9, 11, 16),
    -> PARTITION p4 VALUES IN (10, 13, 15, 18)
    -> );
```

A LIST partitioned table accepts only rows that match a partition.

## HASH Partitioning

- **PARTITION BY HASH** ensures an even distribution of data among partitions.
- Unlike **RANGE** or **LIST**, **HASH** does not require individual partition definition.
- In this “store orders” example, an integer is specified to partition the table into four equal partitions:

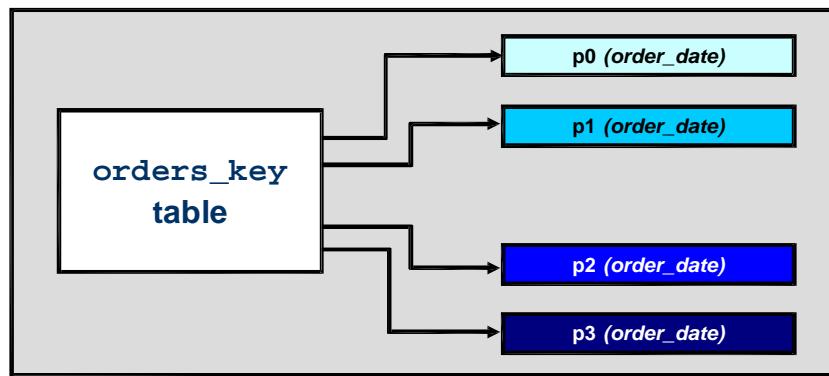


HASH is used primarily to ensure an even distribution of data among a predetermined number of partitions. This distribution is based on an expression that you supply when you create the table. Use the keyword **PARTITIONS** followed by the desired (integral) number of partitions. The following statement creates four partitions based on a hash of the **id** column:

```
mysql> CREATE TABLE orders_hash (
    -> id INT AUTO_INCREMENT PRIMARY KEY, customer_surname VARCHAR( 30 ),
    -> store_id INT, salesperson_id INT, order_date DATE,
    -> note VARCHAR( 500 ) ) ENGINE = InnoDB
    -> PARTITION BY HASH(id) PARTITIONS 4;
```

## KEY Partitioning

- **PARTITION BY KEY** is similar to **HASH** except that the MySQL server uses its own hashing expression.
- Does not require that the partition expression return an integer or **NULL**.
- In this “store orders” example, partitions are specified by date of order:



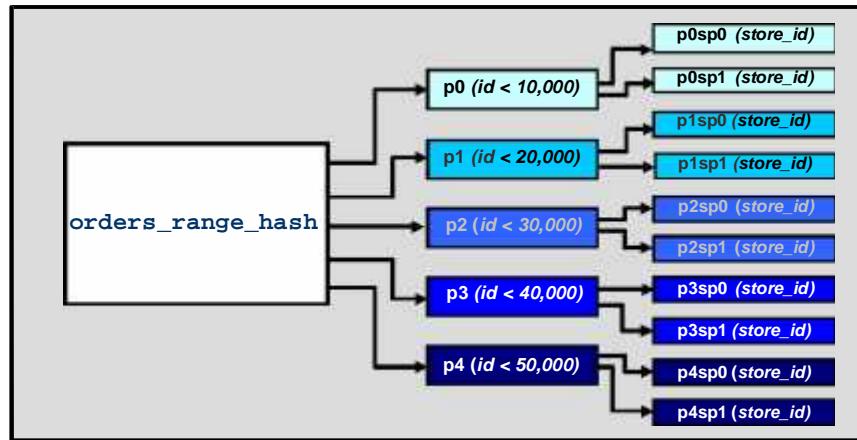
Instead of using a partitioning expression that returns an integer or `NULL`, the expression following `[LINEAR] KEY` consists simply of a list of zero or more column names, with multiple names being separated by commas.

The example in the slide creates the `orders_key` table, which partitions by the date of a placed order:

```
mysql> CREATE TABLE orders_key (
    -> id INT AUTO_INCREMENT, customer_surname VARCHAR(30), store_id INT,
    -> salesperson_id INT, order_date DATE, note VARCHAR(500),
    -> INDEX idx (id) ) ENGINE = InnoDB
    -> PARTITION BY KEY(order_date) PARTITIONS 4;
```

# Subpartitioning

- **RANGE** and **LIST** partitioned tables can be subpartitioned.
- The subpartitions themselves can be **HASH** or **KEY**.
- In this “store orders” database example, based on the `orders_range` table, you can further divide a table so that it splits each partition into two subpartitions:



Subpartitioning (also known as *composite partitioning*) is the further division of each partition in a partitioned table. Subpartitions can use HASH, LINEAR HASH, KEY, or LINEAR KEY partitioning. For instance, you can create a new table (similar to the `orders_range` table) that takes the partitioning a step further with subpartitions:

```
mysql> CREATE TABLE orders_range_hash (
->   ...
-> ) ENGINE = InnoDB
-> PARTITION BY RANGE(id)
->   SUBPARTITION BY HASH(store_id)
->   SUBPARTITIONS 2 (
->     PARTITION p0 VALUES LESS THAN(10000),
->     PARTITION p1 VALUES LESS THAN(20000),
->     PARTITION p2 VALUES LESS THAN(30000),
->     PARTITION p3 VALUES LESS THAN(40000),
->     PARTITION p4 VALUES LESS THAN(50000)
->   );
```

The `orders_range_hash` table is partitioned by RANGE and subpartitioned by HASH. In other words, each of the RANGE partitions is divided into HASH (sub)partitions for a total of  $5 * 2 = 10$  subpartitions.

## Obtaining Partition Information

MySQL provides several methods for determining the partition status of a table:

- **SHOW CREATE TABLE**
  - View the partitioning clauses used in creating a partitioned table.
- **SHOW TABLE STATUS**
  - Determine whether a table is partitioned.
- **INFORMATION\_SCHEMA.PARTITIONS**
  - Query the INFORMATION\_SCHEMA.PARTITIONS table.
- **EXPLAIN PARTITIONS SELECT**
  - Show which partitions are used by a given **SELECT** statement.

## Obtaining Partition Information:

### SHOW CREATE TABLE

Shows the syntax used to create a table, including the

**PARTITION BY** clause:

```
mysql> SHOW CREATE TABLE orders_hash\G
***** 1. row *****
      Table: orders_hash
Create Table: CREATE TABLE `orders_hash` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `customer_surname` varchar(30) DEFAULT NULL,
  `store_id` int(11) DEFAULT NULL,
  `salesperson_id` int(11) DEFAULT NULL,
  `order_date` date DEFAULT NULL,
  `note` varchar(500) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50100 PARTITION BY HASH (id) PARTITIONS 4 */
```

## Obtaining Partition Information: SHOW TABLE STATUS

Determine whether a table is partitioned:

```
mysql> SHOW TABLE STATUS LIKE 'orders_hash'\G
***** 1. row ****
      Name: orders_hash
    Engine: InnoDB
    ...
Create_options: partitioned
    Comment:
```

The SHOW TABLE STATUS statement output is the same as that for non-partitioned tables, except that the Create\_options column contains the string `partitioned`. The Engine column contains the name of the storage engine used by all partitions of the table.

## Obtaining Partition Information: INFORMATION\_SCHEMA

- The **INFORMATION\_SCHEMA.PARTITIONS** table can be queried for partition details.
- Get a list of all database tables and their partitions:

```
mysql> SELECT TABLE_NAME,
      -> GROUP_CONCAT(PARTITION_NAME)
      -> FROM INFORMATION_SCHEMA.PARTITIONS
      -> WHERE TABLE_SCHEMA='orders'
      -> GROUP_BY TABLE_NAME;
+-----+-----+
| table_name | group_concat(partition_name) |
+-----+-----+
| orders_range | p0,p1,p2,p3,p4
| orders_hash | p0,p1,p2,p3
| orders_key | p0,p1,p2,p3
| orders_list | p0,p1,p2,p3,p4
...
...
```

To list the partition names with respective partitioning descriptions for a specific table:

```
mysql> SELECT PARTITION_NAME, PARTITION_DESCRIPTION
      -> FROM INFORMATION_SCHEMA.PARTITIONS
      -> WHERE TABLE_NAME='orders_list'
      -> AND TABLE_SCHEMA='orders';
+-----+-----+
| PARTITION_NAME | PARTITION_DESCRIPTION |
+-----+-----+
| p0             | 1,3,4,17
| p1             | 2,12,14
| p2             | 6,8,20
| p3             | 5,7,9,11,16
| p4             | 10,13,15,18
+-----+-----+
```

## Obtaining Partition Information: **EXPLAIN PARTITIONS**

- Shows how MySQL is processing partitions
- Tells which partitions are being accessed for a query and how they are used
- Can be used only on tables containing row data:

```
EXPLAIN PARTITIONS SELECT * FROM orders_range\G
*****
   id: 1
  select_type: SIMPLE
        table: orders_range
    partitions: p0,p1,p2,p3,p4
       type: ALL
possible_keys: NULL
      key: NULL
     key_len: NULL
       ref: NULL
      rows: 21
     Extra:
```

## Altering a Partition

- Databases are usually dynamic, requiring changes to existing partition settings.
- MySQL allows several types of partition alterations:
  - Add
  - Drop
  - Merge
  - Split
  - Reorganize
  - Redefine
- Simple extensions to the **ALTER TABLE** statement are used to make these alterations.

Although each partition alteration type is important to you as a DBA, most DBAs are more concerned with redefining and dropping partitions than the other types. Therefore, this lesson addresses only these two types.

## Redefining the Partitioning Type

- Change a non-partitioned table into a partitioned table.
- Completely redefine existing partitioning, including type.
- Using **ALTER TABLE** with **PARTITION BY**, change the partition type from **RANGE** to **HASH**:

```
ALTER TABLE orders_range  
PARTITION BY HASH(id) PARTITIONS 4;
```

# Exchanging Partitions

- To exchange a table partition or subpartition with a table.
  - Move any existing rows in the partition or subpartition to the non-partitioned table.
  - Move any existing rows in the non-partitioned table to the table partition or subpartition.
- The table to be exchanged must not be partitioned.
  - It must have the same table structure as the partitioned table.
  - Rows in the non-partitioned table prior to the exchange must lie within the range defined for the partition or subpartition.

```
ALTER TABLE orders_range  
EXCHANGE PARTITION p0  
WITH TABLE orders;
```

When exchanging a partition with a non-partitioned table, the table has the same limitations as a partition. For example, the table must not have a foreign key and must not contain a column that is a foreign key in another table. These and other limitations are covered in a later slide titled “Partitioning Limitations.”

Effects of exchanging partitions include:

- Exchanging a partition does not invoke triggers on either the partitioned table or the exchanged table.
- Any AUTO\_INCREMENT columns in the exchanged table are reset.

# Dropping Partitions

- You can remove one or more partitions.
- This is allowed for **RANGE** or **LIST** tables.
- Using **ALTER TABLE** with **DROP PARTITION**:
  - In this `orders_range` table example, all the order IDs of less than 10,000 have been filled, so the first partition is dropped:

```
ALTER TABLE orders_range DROP PARTITION p0;
```

- The partition scheme is now changed:

| PARTITION_NAME | PARTITION_DESCRIPTION |
|----------------|-----------------------|
| p0             | 10000                 |
| p1             | 20000                 |
| p2             | 30000                 |
| p3             | 40000                 |
| p4             | 50000                 |

Note that dropping a partition is not the same as removing the rows. Although it does have the effect of removing all the data within the partition, the primary intention for using **DROP PARTITION** is to remove the partition designation itself, and to no longer allow data to be funneled into that partition.

You can delete all rows from one or more partitions of a partitioned table using the **ALTER TABLE ... TRUNCATE PARTITION** statement. When you execute this statement, it deletes rows without affecting the structure of the table. The partitions named in the **TRUNCATE PARTITION** clause do not have to be contiguous.

**DROP PARTITION** and **TRUNCATE PARTITION** are similar to **DROP** and **TRUNCATE TABLE** in speed.

## Issues with Using **DROP PARTITION**

- The **DROP** privilege is required to use **DROP PARTITION**.
- Dropping a partition deletes all the data in that partition.
- **DROP PARTITION** does not return the number of rows deleted.
- **DROP PARTITION** changes the table definition.
- Partitions can be dropped in any order.
- Rows inserted into the dropped partitions are handled according to the new partitions.
- Multiple partitions can be dropped in a single statement:

```
ALTER TABLE orders range DROP PARTITION p1, p3;
```

When you drop a partition, you delete all the data that was in that partition, just as if you had run the equivalent **DELETE** statement.

**ALTER TABLE . . . DROP PARTITION** does not return the number of rows that were dropped. If you need this information, use **SELECT COUNT( )** or **DELETE**.

You can drop multiple partitions in a single statement by using the names of the partitions, separated by commas, in the **DROP PARTITION** clause.

Dropping a **LIST** partition prohibits **INSERT** and **UPDATE** operations that would have matched that partition.

Dropping a **RANGE** partition removes the partition and its data. New data from **INSERT** and **UPDATE** operations are stored in the next, greater, partition in the range. If there are no partitions with a range greater than that of the dropped partition, you cannot perform **INSERT** and **UPDATE** operations on data in the dropped partition's range.

The example in the slide shows how to drop partitions **p1** and **p3** from the **orders\_range** table.

# Removing Partitioning

- To remove all partitioning from a table so that it returns to being a non-partitioned table, use an **ALTER TABLE** with **REMOVE PARTITIONING**.
  - This does not delete any table data.
- Example: Return the `orders_range` table to a non-partitioned state:

```
ALTER TABLE orders_range REMOVE PARTITIONING;
```

- The `orders_range` table no longer contains partitions:

| TABLE_NAME   | PARTITION_NAME | PARTITION_DESCRIPTION |
|--------------|----------------|-----------------------|
| orders_range | NULL           | NULL                  |

- **REMOVE PARTITIONING** does a full table copy, like a normal **ALTER TABLE ALGORITHM=COPY**.

After the partitioning is removed, the table definition no longer includes the `PARTITION BY` portion. Also there are no longer any partitions listed if you attempt to query the partitions in the `INFORMATION_SCHEMA.PARTITIONS` table, as shown in the example in the slide.

## Performance Effects of Altering a Partition

- Depending on the number of partitions, creating partitioned tables is slightly slower than creating non-partitioned tables.
- Partitioning operation processing speed comparisons:
  - **DROP PARTITION** is much faster than **DELETE** for large transactional tables.
  - **ADD PARTITION** is fairly quick on **RANGE** and **LIST** tables.
  - For **ADD PARTITION** on **KEY** or **HASH** tables, speed depends on the number of rows already stored.
    - It takes longer to add new partitions when there is more data.
  - **COALESCE PARTITION**, **REORGANIZE PARTITION**, and **PARTITION BY** can be slow when run on very large tables.
- During such operations, the overhead of hardware I/O is much higher than that of the partitioning engine.

**ADD PARTITION** on **KEY/HASH** redistributes all rows to the new number of partitions, effectively working as a full table copy. The same result occurs with **COALESCE PARTITION** and **PARTITION BY**.

**ADD/COALESCE PARTITION** for **LINEAR HASH/KEY** partitions only splits/merges the affected number of partitions.

**REORGANIZE PARTITION** depends on the size of the partitions to reorganize.

# **Quiz**

The types of partitioning include:

- a. LINEAR, COLUMNS, B-TREE, HASH
- b. [LINEAR] LIST, [LINEAR] RANGE, HASH  
[COLUMNS], KEY [COLUMNS]
- c. LIST, RANGE, HASH, KEY
- d. RANGE [COLUMNS], LIST [COLUMNS], [LINEAR]  
HASH, [LINEAR] KEY

# Partitioning: Storage Engine Features

- Partitions are stored in files in the same location as the InnoDB table.
  - You can provide a **DATA DIRECTORY** option to relocate the partition.
- Each partition has its own file in the data directory:  
**<table\_name>#P#<partition\_name>.ibd**
  - With `innodb_file_per_table` disabled, partitions are stored in the shared tablespace.
- All data and indexes are partitioned.
  - You cannot partition only data or only indexes.
- Partitioning is available in other storage engines.
  - Not available in MERGE, FEDERATED, CSV

Each PARTITION clause can include a [ STORAGE ] ENGINE option. This has no effect. Each partition uses the same storage engine as that used by the table as a whole.

Partitioning applies to all data and indexes of a table. Note that you cannot partition only the data or only the indexes.

Partitions are stored in their own files in the data directory by default. Use DATA DIRECTORY to specify an alternative partition location:

```
CREATE TABLE entries (id INT, entered DATE)
PARTITION BY RANGE(YEAR(entered)) (
    PARTITION p0 VALUES LESS THAN (2000) DATA DIRECTORY = '/data/p0',
    PARTITION p1 VALUES LESS THAN MAXVALUE DATA DIRECTORY = '/data/p1'
);
```

Partitioning is implemented as a storage engine. A partitioned table uses a combination of the partitioning storage engine and the backing storage engine (for example, InnoDB). Other storage engines can also use partitioning; however, the MERGE, CSV, and FEDERATED storage engines cannot use partitioning.

# Partitioning and Locking

- MySQL tables are locked as part of the server code.
  - Locking is handled by the table storage engine during each statement.
  - Each storage engine deals with locks differently.
- There is one storage engine instance per partition.
  - Each non-pruned partition/storage engine instance is asked to be locked.
- When a partitioned table is locked, the non-pruned partitions are locked.
  - The lock is held only during the execution of a statement.
- **ALTER...PARTITION** statements normally take a write lock on the table.

When you execute an `ALTER...PARTITION` operation on a table, it takes a *write lock* on the table. Reads from such tables are relatively unaffected; pending `INSERT` and `UPDATE` operations are performed as soon as the partitioning operation has completed.

When you execute an `ALTER TABLE t <CMD> PARTITION` type statement, it requires a write lock on the table during the copy phase, unless the command supports online operations with `ALGORITHM=INPLACE`. When the copy phase is done, it needs exclusive access to the table to change the table definition.

# Partitioning Limitations

- General
  - The maximum number of partitions per table is 8192.
  - Spatial types are not supported.
  - Temporary tables cannot be partitioned.
  - It is not possible to partition log tables.
- Foreign keys and indexes
  - Foreign keys are not supported.
  - FULLTEXT indexes are not supported.
  - No global indexes: Each partition has its own indexes.
- Subpartitioning is possible only:
  - When partitioning by **RANGE** and **LIST**
  - By **LINEAR HASH** or **LINEAR KEY**

## Partition Expression Limitations

- Expressions used for **RANGE**, **LIST**, and **HASH** partitions must evaluate as an integer.
  - **RANGE COLUMNS** and **LIST COLUMNS** allow a wider range of data types.
- You cannot use **TEXT** or **BLOB** in partitioning expressions.
- UDFs, stored functions, variables, some operators, and some built-in functions are not allowed.
  - Operators: **|**, **&**, **^**, **<<**, **>>**, **~**
- SQL modes should not be changed after table creation.
- Subqueries are not supported in partitioning expressions.
- All columns used in the partitioning expression must be part of all of the table's unique indexes.

The **RANGE COLUMNS** and **LIST COLUMNS** partition types do not require integer columns or expressions. You can use multiple columns in partitioning keys. The columns can be any of the integer types, DATE or DATETIME, CHAR, VARCHAR, BINARY, or VARBINARY. For further information about COLUMNS partitioning, see the following page:

<http://dev.mysql.com/doc/mysql/en/partitioning-columns.html>

The unique key that must reference the columns in the partitioning expression includes the table's primary key, because it is by definition a unique key. This is because the indexes are partitioned together with the data. To keep the "uniqueness," two rows with the same unique key must go to the same partition; otherwise, they violate the unique key restriction.

## **Summary**

In this lesson, you should have learned how to:

- Define partitioning and its particular use in MySQL
- Determine server partitioning support
- List the reasons for using partitioning
- Explain the types of partitioning
- Create partitioned tables
- Describe subpartitioning
- Obtain partitioning metadata
- Use partitioning to improve performance
- Explain storage engine implementation of partitioning
- Explain how to lock a partitioned table
- Describe partitioning limitations

## **Chapter 6**

### User Management

# Objectives

After completing this lesson, you should be able to:

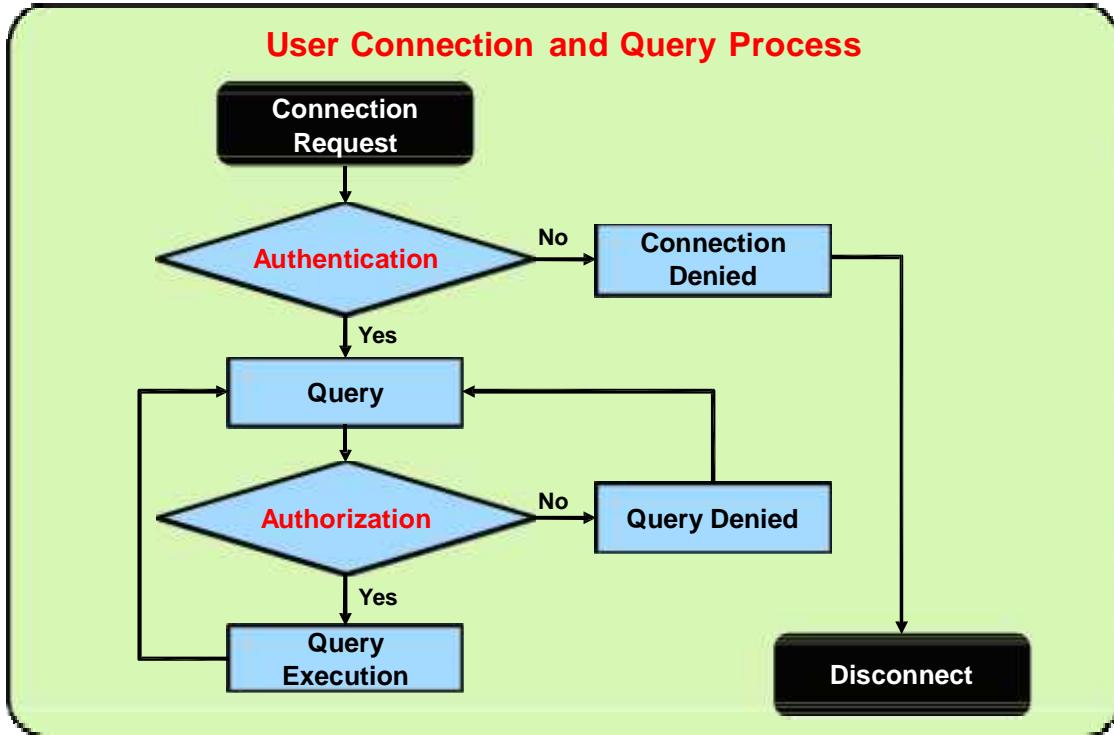
- Describe the user connection and query process
- List requirements for user authentication
- Create, modify, and drop user accounts
- Configure authentication plugins
- List requirements for user authorization
- Describe the levels of access privileges for users
- List the types of privileges
- Grant, modify, and revoke user privileges
- Use **SHOW PROCESSLIST** to display running threads
- Disable server client access control
- Set account resource limits

## **Importance of User Management**

Managing users in MySQL gives you the ability to control what users can and cannot do.

- Create user accounts with different privileges that are appropriate to their function.
- Avoid using the `root` account.
  - Constrain compromised applications.
  - Protect against mistakes during routine maintenance.
- Ensure data integrity by proper assignment of individual user privileges.
  - Permit authorized users to do their work.
  - Prevent unauthorized users from accessing data beyond their privileges.

# User Account Verification



When you connect to a MySQL server and execute a query, it authenticates you and authorizes your activity.

- **Authentication:** Verifies the user's identity. This is the first stage of access control. You must successfully authenticate each time you connect. If you fail to authenticate, your connection fails and your client disconnects.
- **Authorization:** Verifies the user's privileges. This second stage of access control takes place for each request on an active connection on which authentication has succeeded. For every request, MySQL determines what operation you want to perform, and then checks whether you have sufficient privileges to do so.

## Viewing User Account Settings

- Query the mysql database to view user identification info:

```
mysql> SELECT user, host, password
-> FROM mysql.user WHERE user='root';
+-----+-----+-----+
| user | host   | password          |
+-----+-----+-----+
| root | localhost | *2447D497B9A6A15F2776055CB2D1E9F86758182F |
| root | 127.0.0.1 | *2447D497B9A6A15F2776055CB2D1E9F86758182F |
| root | ::1     | *2447D497B9A6A15F2776055CB2D1E9F86758182F |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

- View all user info, including privileges:

```
mysql> SELECT * FROM mysql.user\G
***** 1. row *****
...
      Select_priv: Y
      Insert_priv: Y
      Update_priv: Y
      Delete_priv: Y
      ...
...
```

The mysql database contains the information for all user accounts on the server. To view this information, run the **SELECT** statements shown in the slide.

The value **Y** in a **\*\_priv** field indicates that the privilege is enabled.

The **root** account has full access. All of its privilege columns have the value **Y**.

As well as privileges, the user table contains other information that is useful in the authentication process. For example, you can see in the following output that the `tester` user:

- Has a password (visible in encrypted form in the `Password` column), and this password is not expired (indicated by the `N` in the `password_expired` column)
- Has no defined resource limits (indicated by the `0s` in the `max_*` columns)
- Does not have any SSL or x509 settings (indicated by the blank values in the `ssl_*` and `x509_*` columns)
- Uses the `mysql_native_password` plugin to authenticate (The plugin name is listed in the `plugin` column.)

```
***** 1. row *****
      Host: localhost
      User: testuser
      Password: *14E65567ABDB5135D0CFD9A70B3032C179A49EE7
      Select_priv: Y
      Insert_priv: N
      ...
      Trigger_priv: N
Create_tablespace_priv: N
      ssl_type:
      ssl_cipher:
      x509_issuer:
      x509_subject:
      max_questions: 0
      max_updates: 0
      max_connections: 0
      max_user_connections: 0
      plugin: mysql_native_password
authentication_string:
      password_expired: N
```

These features are described throughout the remainder of this lesson. Connecting securely using SSL is explored further in the lesson titled “Security” later in this course.

## Native Authentication

- During connection using the `mysql_native_password` plugin, MySQL uses the following to authenticate an account:
  - Username
  - Password
  - Client host
- When specifying host names, keep the proper perspective in mind:
  - Specify the server's host name when connecting using a client.
  - Specify the client's host name when adding a user to the server.

When you connect to a MySQL server using the native password authentication plugin (the default authentication mechanism), it matches the username that you specified, the host from which you are connecting, and your password against rows in the `mysql.user` table to determine whether you can connect and perform actions.

To connect to the local server using the `mysql` client, specify the username and password for the account that you want to use:

```
shell> mysql -u<username> -p<password>
```

Note that the host name associated with your user in the `mysql.user` table refers to the name of the host from which you are connecting, not to the server host.

To connect to a server that is not installed on your client's local host, provide the host name of the server to which you are connecting:

```
shell> mysql -u<username> -p<password> -h<server_host>
```

## Creating a User Account

- Provide a user and host for each user account.
- For example, use the **CREATE USER... IDENTIFIED BY** statement to build an account:
  - For a user named jim
  - To connect from localhost
  - Using the password Abc123

```
CREATE USER 'jim'@'localhost' IDENTIFIED BY 'Abc123';
```

- Avoid possible security risks when creating accounts:
  - Do not create accounts without a password.
  - Do not create anonymous accounts.
  - Where possible, avoid wildcards when you specify account host names.

An account name consists of a username and the name of the client host from which the user must connect to the server. Account names have the format '*user\_name*' '@' '*host\_name*'. Usernames can be up to 16 characters long. You must use single quotation marks around usernames and host names if these contain special characters, such as dashes. If a value is valid as an unquoted identifier, the quotes are optional. However, you can always use quotes.

# Host Name Patterns

Examples of permissible host name formats:

- Host name: **localhost**
- Qualified host name: '**hostname.example.com**'
- IP number: **192.168.9.78**
- IP address: **10.0.0.0/255.255.255.0**
- Pattern or wildcard: **%** or **\_**

Use a host pattern containing the % or \_ wildcard characters to set up an account that enables the user to connect from any host in an entire domain or subnet.

If you omit the host part of an account name when writing an account management statement, MySQL assumes a host name of %.

A host value of %.example.com matches any host in the example.com domain. A host value of 192.168.% matches any host in the 192.168 subnet. A host value of % matches any host, permitting the user to connect from any host.

Use an IP address with a subnet mask to enable the user to connect from any host with an address within that subnet. For example, a value of 10.0.0.0/255.255.255.0 matches any host with 10.0.0 in the first 24 bits of its IP address.

Avoid using wildcards in host names except where it is strictly necessary and properly audited to avoid abuse or accidental exposure. Run periodic checks as follows:

```
mysql> SELECT User, Host FROM mysql.user WHERE Host LIKE '%\%%';
```

Username and host name examples:

- john@10.20.30.40
- john@'10.20.30.%'
- john@'%ourdomain.com'
- john@'10.20.30.0/255.255.255.0'

To specify an anonymous-user account (that is, an account that matches any username), specify an empty string for the username part of the account name:

```
mysql> CREATE USER ''@'localhost';
```

Avoid creating anonymous accounts, especially ones that have no password (as in the preceding example). This helps avoid security risks that would come from opening up access to the MySQL installation.

If a host matches two or more patterns, MySQL chooses the most specific pattern.

## Setting the Account Password

- There are several ways to set a MySQL user password:
  - **CREATE USER...IDENTIFIED BY**
  - **GRANT...IDENTIFIED BY**
  - **SET PASSWORD**
  - **mysqladmin password**
  - **UPDATE grant tables (not recommended)**
- The **SET PASSWORD** statement is the most common method for setting or changing an account password, as in this example:

```
SET PASSWORD FOR 'jim'@'localhost'  
= PASSWORD('NewPass');
```

The most common way to change an existing account's password without changing any of its privileges is to use the **SET PASSWORD** statement. For example, to set the password for `jim` on the local host to `NewPass`, use the following statement:

```
mysql> SET PASSWORD FOR jim@localhost = PASSWORD('NewPass');
```

If you are logged in as a non-root user and your user does not have the **UPDATE** privilege for the `mysql` database, you can change only your own password. Do this by using the **SET PASSWORD** statement without the **FOR** clause:

```
mysql> SET PASSWORD = PASSWORD('NewPass');
```

When using **SET PASSWORD**, use the **PASSWORD()** function to encrypt the password. Note that the **CREATE USER** statement automatically encrypts the password that you provide, so you do not need to use the **PASSWORD()** function when creating a user with **CREATE USER**.

Use the following `mysqladmin` commands to set passwords from the shell: `shell>`

```
mysqladmin -u root -pCurrPass password 'NewPass' shell>  
mysqladmin -u root -pCurrPass -h host_name password  
'NewPassword'
```

In the preceding examples, `NewPass` represents the new root password and `host_name` is the name of the host from which the root account accesses the MySQL server.

# Confirming Passwords

Assign strong, unique passwords to all user accounts.

- Avoid passwords that can be easily guessed.
- Use the following **SELECT** statement to list any accounts without passwords:

```
SELECT Host, User FROM mysql.user  
WHERE Password = '';
```

- Identify duplicate passwords:

```
SELECT User FROM mysql.user GROUP BY password  
HAVING count(user)>1;
```

- Expire passwords:

```
ALTER USER jim@localhost PASSWORD EXPIRE;
```

To issue the SELECT statements in the slide, you must connect with a user account with SELECT privileges on the mysql schema or mysql.user table.

You can have multiple accounts that apply to a specific username. For example, if the user `jim` logs in from two locations and you set up accounts for each location, such as `jim@localhost` and `jim@'192.168.14.38'`, both accounts identified as `jim` might have the same password.

You can expire a user's password with the ALTER USER...PASSWORD EXPIRE statement. If your password expires, you must change your password using a SET PASSWORD statement the next time you log in. All statements you execute that do not start with SET return an error until you change your password, as in this example:

```
mysql> SELECT * FROM City WHERE 1=2;  
ERROR 1820 (HY000): You must SET PASSWORD before executing this  
statement  
mysql> SET PASSWORD = PASSWORD('new_pwd');  
Query OK, 0 rows affected (0.01 sec)  
mysql> SELECT * FROM City WHERE 1=2;  
Empty set (0.00 sec)
```

# Manipulating User Accounts

- Use the **RENAME USER** statement to rename a user account:

```
RENAME USER 'jim'@'localhost' TO 'james'@'localhost';
```

- Changes the account name of an existing account
- Changes either the username or host name parts of the account name, or both

- Use the **DROP USER** statement to remove a user account:

```
DROP USER 'jim'@'localhost';
```

- Revokes all privileges for an existing account and then removes the account
  - Deletes all records for the account from any grant table in which they exist
- Rename or remove users when their access requirements change.

# Pluggable Authentication

MySQL supports a number of authentication mechanisms that are available through pluggable authentication.

- Plugins are built-in or available as external libraries.
- Default server-side plugins are built-in, always available, and include:
  - **mysql\_native\_password**: This is the default mechanism, as described in the preceding slides.
  - **mysql\_old\_password**: This plugin implements authentication as used before MySQL4.1.1.
  - **sha256\_password**: This plugin enables SHA-256 hashing of passwords.

MySQL uses a number of algorithms to encrypt passwords stored in the user table:

- The `mysql_native_password` plugin implements the standard password format, a 41-byte-wide hash.
- The `mysql_old_password` plugin implements an older format that is less secure, being 16 bytes wide.
- The `sha256_password` plugin implements the SHA-256 hashing algorithm widely used in secure computing.

The value of the `old_passwords` system variable specifies the algorithm that the `PASSWORD( )` function uses to create passwords, as follows:

- 0: The standard algorithm, as used since MySQL 4.1.1
- 1: The old algorithm, as used before MySQL 4.1.1
- 2: The SHA-256 algorithm

Start the server with the `default-authentication-plugin` option set to `sha256_password` to use SHA-256 passwords for all new users, or use `CREATE USER` with the `IDENTIFIED WITH sha256_password` clause to specify SHA-256 passwords for a specific user.

## Client-Side Cleartext Authentication Plugin

The MySQL client library includes a built-in Cleartext Authentication plugin, `mysql_clear_password`. The plugin is:

- Used to send a plain text password to the server
  - The password is usually hashed.
- Enabled by:
  - The `LIBMYSQL_ENABLE_CLEARTEXT_PLUGIN` environment variable
  - Specifying `--enable-cleartext-plugin` when running MySQL client applications such as `mysql` and `mysqladmin`
  - The `MYSQL_ENABLE_CLEARTEXT_PLUGIN` option of the `mysql_options()` C API function

Some authentication methods, such as PAM (Pluggable Authentication Modules) authentication, require the client to send a plain-text password to the server so that the server can process the password in its normal form. The `mysql_clear_password` plugin enables this behavior.

# Loadable Authentication Plugins

- **test\_plugin\_server**: Implements native and old password authentication
  - This plugin uses the `auth_test_plugin.so` file.
- **auth\_socket**: Allows only MySQL users who are logged in via a UNIX socket from a UNIX account with the same name
  - This plugin uses the `auth_socket.so` file.
- **authentication\_pam**: Allows you to log in using an external authentication mechanism
  - This plugin uses the `authentication_pam.so` file.

To load one of these plugins, start the server with the `plugin-load` option set to the plugin's file name.

In addition to default built-in plugins, MySQL provides several loadable plugins:

- The Test Authentication plugin (`test_plugin_server`) authenticates using native or old password authentication, and is intended for testing and development purposes.
- The Socket Peer-Credential (`auth_socket`) plugin allows users to connect via the UNIX socket file only if their Linux username matches their MySQL account.
- The PAM authentication plugin (`authentication_pam`) is an Enterprise Edition plugin that allows you to log in using an external authentication mechanism. MySQL does not store your password, but uses the UNIX PAM (Pluggable Authentication Modules) mechanism to transmit the client's provided username and password for authentication by the operating system.

You can develop your own authentication plugins. The Test Authentication plugin is intended for use by developers to create their own plugins; its source code is available as part of the MySQL source code distribution.

Load a loadable authentication plugin by starting the server with the `plugin-load` option at the command line or in the `my.cnf` file, as in the following example:

```
[mysqld]
plugin-load=authentication_pam.so
```

# PAM Authentication Plugin

- The PAM Authentication plugin is an Enterprise Edition plugin that authenticates MySQL accounts against the operating system.
- PAM defines services that configure authentication.
  - These are stored in /etc/pam.d.
- The plugin authenticates against:
  - Operating system users and groups
  - External authentication such as LDAP

To create MySQL users that authenticate with PAM, do the following:

```
CREATE USER user@host  
    IDENTIFIED WITH authentication_pam  
    AS 'pam_service, os_group=mysql_user'
```

PAM looks in /etc/pam.d for services that it authenticates. For example, to create a PAM service called mysql-pam, create the file /etc/pam.d/mysql-pam with the following content:

```
#%PAM-1.0  
auth      include  password-auth  
account   include  password-auth
```

In addition to MySQL authentication, PAM integrates with other authentication methods including LDAP and Active Directory, so you can use PAM to authenticate many services (including MySQL) against a single store in your network.

To create a MySQL user that maps directly to an operating system user, use a statement such as the following:

```
CREATE USER bob@localhost  
    IDENTIFIED WITH authentication_pam AS 'mysql-pam';
```

When bob logs in, MySQL passes the username and password that it receives from the client to PAM, which authenticates against the operating system. The client must send the password in clear text. Enable the client-side Cleartext Authentication plugin to serve this purpose:

```
shell> mysql --enable-cleartext-plugin -ubob -p  
Enter password: bob's_OS_password
```

To enable group-based logins with the PAM Authentication plugin, create a PAM-enabled anonymous proxy account that matches no users, but specifies a set of mappings from operating system group to MySQL user:

```
CREATE USER ''@'' IDENTIFIED WITH authentication_pam  
    AS 'mysql-pam, sales=m_sales, finance=m_finance';
```

The preceding example assumes that you have `sales` and `finance` operating system groups and `m_sales` and `m_finance` MySQL users.

You must then grant the `PROXY` privilege to the anonymous proxy account, giving it rights to log in as the `m_sales` and `m_finance` MySQL users:

```
GRANT PROXY ON m_sales@localhost TO ''@'';  
GRANT PROXY ON m_finance@localhost TO ''@'';
```

Users who are members of the `sales` and `finance` groups can now provide their operating system credentials at the `mysql` command-line prompt, which logs them in as the `m_sales` or `m_finance` MySQL users, respectively, giving them all of the privileges granted to those accounts. For example, if `peter` is a member of the `sales` group, he could log in as follows:

```
shell> mysql --enable-cleartext-plugin -upeter -p  
Enter password: peter's_OS_password  
Welcome to the MySQL monitor. Commands end with ; or \g.  
...  
mysql> SELECT CURRENT_USER();  
+-----+  
| CURRENT_USER() |  
+-----+  
| m_sales@localhost |  
+-----+  
1 row in set (0.01 sec)
```

# Password Validation Plugin

The `validate_password` plugin tests the strength of supplied passwords to improve security.

- For statements that assign a cleartext password, the password is checked against the value of the `validate_password_policy` variable.
  - These include `CREATE USER`, `GRANT`, `SET PASSWORD` and arguments to the `PASSWORD()` function.
  - Possible values for `validate_password_policy` are `LOW`, `MEDIUM` or `STRONG`
- You can check the strength of a password with the `VALIDATE_PASSWORD_STRENGTH()` function
  - Returns an integer in the range 0 (weak) to 100 (strong)

The `validate_password_policy` variable enforces three levels of validation:

- `LOW`: Passwords must be of a minimum length, as determined by the value of the `validate_password_length` variable.
- `MEDIUM`: In addition to the requirements of `LOW`, passwords must also contain a minimum number of numeric characters, mixed case characters, and special (nonalphanumeric) characters as specified by the `validate_password_number_count`, `validate_password_mixed_case_count` and `validate_password_special_char_count` variables respectively.
- `STRONG`: In addition to the requirements of `MEDIUM`, password substrings of four or more characters must not match words in the dictionary file specified in the `validate_password_dictionary` variable.

The default values of the validate\_password variables are shown below:

```
mysql> show variables like 'val%';
```

| Variable_name                        | Value  |
|--------------------------------------|--------|
| validate_password_dictionary_file    |        |
| validate_password_length             | 8      |
| validate_password_mixed_case_count   | 1      |
| validate_password_number_count       | 1      |
| validate_password_policy             | MEDIUM |
| validate_password_special_char_count | 1      |

# Authorization

- After a user authenticates, MySQL asks the following questions to verify account privileges:
  - Who is the user?
  - What privileges does the user have?
  - Where do these privileges apply?
- You must set up proper accounts and privileges for authorization to work.

The primary function of the MySQL authorization system is to associate an authenticated user with privileges on a database, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. The authorization system's functionality includes the ability to have anonymous users, and to enable specific functions such as `LOAD DATA INFILE` and various administrative operations. This authorization ensures that users can perform only the operations for which they have been granted appropriate privileges.

## Determining Appropriate User Privileges

- Grant privileges at the correct level of access:
  - Global
  - Database
  - Table
  - Column
  - Stored routine

- Grant privileges to users according to their access requirements:
  - **Read-only users:**
    - Global-, database-, or table-level privileges such as `SELECT`
  - **Users who modify databases:**
    - Global-, database-, or table-level privileges such as `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ALTER`, and `DROP`
  - **Administrative users:**
    - Global-level privileges such as `FILE`, `PROCESS`, `SHUTDOWN`, and `SUPER`

You can grant several types of privileges to a MySQL account at different levels: globally, or for particular databases, tables, or columns. For example, you can give a user the ability to select from any table in any database by granting the user the `SELECT` privilege at the global level.

You can give an account complete control over a specific database without having any permissions on other databases. The account can then create the database, create tables and other database objects, select from the tables, and add, delete, or update new records.

# Granting Administrative Privileges

The following global privileges apply to administrative users:

- **FILE**: Allows users to instruct the MySQL server to read and write files in the server host file system
- **PROCESS**: Allows users to use the **SHOW PROCESSLIST** statement to see all statements that clients are executing
- **SUPER**: Allows users to kill other client connections or change the runtime configuration of the server
- **ALL**: Grants all privileges except the ability to grant privileges to other users

Grant administrative privileges sparingly, because they can be abused by malicious or careless users.

The SUPER administrative privilege provides users with the ability to perform additional tasks, including setting global variables and terminating client connections.

There are also some special privilege specifiers:

- Use **ALL** and **ALL PRIVILEGES** for granting all privileges except the ability to give privileges to other accounts. Use **GRANT ALL ... WITH GRANT OPTION** to grant all privileges including the ability to give privileges to other accounts.
- Use **USAGE** to grant the ability to connect to the server. This privilege creates a record in the `user` table for the account, but without any privileges. The account can then be used to access the server for limited purposes, such as issuing **SHOW VARIABLES** or **SHOW STATUS** statements. The account cannot be used to access database contents such as tables, although such privileges can be granted at a later time.

Other administrative privileges include **CREATE USER**, **CREATE TEMPORARY TABLES**, **SHOW DATABASES**, **LOCK TABLES**, **RELOAD**, and **SHUTDOWN**. Administrative privileges, including those in the slide, can be used to compromise security, access privileged data, or perform denial-of-service attacks on a server. Ensure that you grant these privileges only to appropriate accounts.

For more information about granting MySQL privileges, see the *MySQL Reference Manual*:  
<http://dev.mysql.com/doc/mysql/en/privileges-provided.html>

## GRANT Statement

- The **GRANT** statement creates a new account or modifies an existing account.
- **GRANT** syntax:

```
GRANT SELECT ON world_innodb.* TO  
    'kari'@'localhost' IDENTIFIED BY 'Abc123';
```

- Clauses of the statement:
  - Privileges to be granted
  - Privilege level:
    - Global: \*.\*
    - Database: <db\_name>.\*
    - Table: <db\_name>.<table\_name>
    - Stored routine: <db\_name>.<routine\_name>
  - Account to which you are granting the privilege
  - An optional password

In the example in the slide, the statement grants the `SELECT` privilege for all tables in the `world_innodb` database to a user named `kari`, who must connect from the local host and use a password of `Abc123`.

The clauses of the GRANT statement have the following effects:

- **GRANT keyword:** Specifies one or more privilege names indicating which privileges you are granting. Privilege names are not case-sensitive. To list multiple privileges, separate them by commas.
- **ON clause:** Specifies the level of the privileges that you are granting
- **TO clause:** Specifies the account to which you are granting the privileges. If the account does not already exist, the statement creates it.
- **IDENTIFIED BY clause:** (Optional) Assigns the specified password to the account. If the account already exists, the password replaces any old one.

Omitting the `IDENTIFIED BY` clause has the following effect:

- If the account in the `TO` clause exists, its password remains unchanged.
- If the account in the `TO` clause does not exist, it is created with a blank password.

As a security measure, enable the `NO_AUTO_CREATE_USER` SQL mode to prevent the `GRANT` statement from creating new accounts when you do not specify an `IDENTIFIED BY` clause.

## Display GRANT Privileges

- Show general account privileges with the **SHOW GRANTS** statement:

```
SHOW GRANTS;  
SHOW GRANTS FOR CURRENT_USER();
```

- Specify an account name:

```
mysql> SHOW GRANTS FOR  
'kari'@'myhost.example.com';  
+-----+  
| Grants for kari@myhost.example.com |  
+-----+  
| GRANT FILE ON *.* TO 'kari'@'myhost.example.com'  
| GRANT SELECT ON `world_innodb`.* TO 'kari'@'myhost.example.com'  
| IDENTIFIED BY PASSWORD  
| '*E74858DB86EBA20BC33D0AEC8AE8A8108C56B17FA'  
+-----+
```

- Passwords are stored and displayed in an encrypted form.

SHOW GRANTS displays the statements that re-create the privileges for the specified user. It shows privileges only for the exact account specified in the statement. For example, the example in the slide shows privileges only for `kari@myhost.example.com`, not for `kari@%`.

The output for the third statement displayed in the slide consists of two GRANT statements. Their ON clauses display privileges at the global and database levels, respectively.

If the account has a password, SHOW GRANTS displays an IDENTIFIED BY PASSWORD clause at the end of the GRANT statement; this clause lists the account's global privileges. The word PASSWORD after IDENTIFIED BY indicates that the password value shown is the encrypted value stored in the user table, not the actual password. Because the password is stored using one-way encryption, MySQL has no way to display the unencrypted password.

If the account can grant some or all of its privileges to other accounts, the output displays WITH GRANT OPTION at the end of each GRANT statement to which it applies.

## User Privilege Restrictions

- You cannot explicitly deny access to a specific user.
- You cannot associate a password with a specific object such as a database, table, or routine.

## Grant Tables

- MySQL server reads the grant tables from the `mysql` database into memory at startup, and bases all access control decisions on those tables.
- Tables correspond to privilege levels:

| Privilege Level/<br>Table | Contents and Privileges                                |
|---------------------------|--|
| <code>user</code>         | Contains a record for each account known to the server |
| <code>db</code>           | Database-specific privileges                           |
| <code>tables_priv</code>  | Table-specific privileges                              |
| <code>columns_priv</code> | Column-specific privileges                             |
| <code>procs_priv</code>   | Stored procedures and functions privileges             |

- The tables indicate that the validity and privileges of each account and each level are progressively more secure.

Grant tables contain information to indicate what the valid accounts are and the privileges held at each access level by each account.

The `user` table contains a record for each account known to the server, as well as its global privileges. It also indicates other information about the account, such as:

- Any resource limits that it is subject to
- Whether client connections that use the account must be made over a secure connection using SSL

Every account must have a `user` table record; the server determines whether to accept or reject each connection attempt by reading the contents of that table.

Each account also has records in the other grant tables if it has privileges at a level other than the global level.

## Use of Grant Tables

- Each grant table has `host` and `user` columns to identify the accounts to which its records apply.
  - During connection attempts, the server determines whether a client can connect.
  - After connection, the server determines access privileges for each statement.
- The MySQL installation process creates the grant tables.
  - Grant tables use the MyISAM storage engine.
  - MyISAM is guaranteed to be available.

The server determines whether a client can connect based on the `Host`, `User`, and `Password` columns of the `user` table. To connect successfully, MySQL must match a record in the `user` table to the host from which the client connects, the username given by the client, and the password listed in the matching record.

After a client connects, MySQL checks the access privileges for each statement by matching the account's identity to the `Host` and `User` columns of the privilege tables.

- Privileges in each row of the `user` table apply globally to the account identified by its `Host` and `User` columns.
- The privileges in the matching records of the `db`, `tables_priv`, `columns_priv`, and `procs_priv` tables apply at the level identified by the name of the specific privilege table.

For example, privileges in a `db` table record apply to the database named in the record, but not to other databases.

## Effecting Privilege Changes

- MySQL maintains in-memory copies of grant tables to avoid the overhead of accessing on-disk tables.
  - Avoid modifying a user account directly in the grant tables.
  - If you modify the grant tables directly, reload the tables explicitly by issuing a **FLUSH PRIVILEGES** statement.
- Account modification statements such as GRANT, REVOKE, SET PASSWORD, and RENAME USER apply changes to both the grant tables and the in-memory table copies.
- Changes to global privileges and passwords apply only to subsequent connections of that account.
- Changes to database-level privileges apply after the client's next `USE db_name` statement.
- Changes to table and routine privileges apply immediately.

The server reads the grant tables into memory during its startup sequence and uses the in-memory copies to check client access.

The server refreshes its in-memory copies of the grant tables under the following conditions:

- You modify a user account by issuing an account management statement such as CREATE USER, GRANT, REVOKE, or SET PASSWORD.
- You reload the tables explicitly by issuing a **FLUSH PRIVILEGES** statement or by executing a `mysqladmin flush-privileges` or `mysqladmin reload` command.

Avoid making changes directly to the grant tables for the following reasons:

- The syntax of account management statements is designed to be clear and straightforward.
- If you make a mistake in an account management statement, the statement fails and does not change any settings.
- If you make a mistake when changing grant tables directly, you can lock all users out of the system.

# Revoking Account Privileges

- Use the **REVOKE** statement to revoke specific SQL statement privileges:

```
REVOKE DELETE, INSERT, UPDATE ON world_innodb.*  
FROM 'Amon'@'localhost';
```

- Revoke the privilege to grant privileges to other users:

```
REVOKE GRANT OPTION ON world_innodb.*  
FROM 'Jan'@'localhost';
```

- Revoke all privileges, including granting privileges to others:

```
REVOKE ALL PRIVILEGES, GRANT OPTION  
FROM 'Sasha'@'localhost';
```

- Use the **SHOW GRANTS** statement before issuing **REVOKE** to determine which privileges to revoke, and then again afterward to confirm the result.

Use the **REVOKE** statement to revoke privileges from an account. You may choose to revoke privileges for various reasons, such as a reduction in a user's required access. The **REVOKE** statement's syntax has the following clauses:

- **REVOKE keyword:** Specifies the list of privileges to be revoked
- **ON clause:** Indicates the level at which privileges are to be revoked
- **FROM clause:** Specifies the account name

The examples in the slide assume three users, all on `localhost`:

- Assume that Amon has `SELECT`, `DELETE`, `INSERT`, and `UPDATE` privileges on the `world_innodb` database, but you want to change the account so that he has `SELECT` access only. The first example revokes the privileges that enable him to make changes.
- The second example revokes Jan's ability to grant to other users any privileges that he holds for the `world_innodb` database, by revoking the `GRANT OPTION` privilege from his account.
- The third example revokes all privileges held by Sasha's account (at any level), by revoking `ALL PRIVILEGES` and `GRANT OPTION` from her account.

## SHOW PROCESSLIST

- **SHOW PROCESSLIST** shows you which process threads are running.
- **SHOW PROCESSLIST** produces the following columns:
  - **Id**: Connection identifier
  - **User**: MySQL user who issued the statement
  - **Host**: Host name of the client issuing the statement
  - **db**: Default database selected; otherwise NULL
  - **Command**: Type of command that the thread is executing
  - **Time**: Time (in seconds) that the thread has been in its current state
  - **State**: Action, event, or state indicating what the thread is doing
  - **Info**: Statement that the thread is executing; otherwise NULL
- The **PROCESS** privilege permits you to see all threads.

You can also get thread information from the `INFORMATION_SCHEMA.PROCESSLIST` table or the `mysqladmin processlist` command. If you do not have the `PROCESS` privilege, you can view only your own threads. That is, you can view only those threads associated with the MySQL account that you are using. If you do not use the `FULL` keyword, only the first 100 characters of each statement are shown in the `Info` field.

Using the `SHOW FULL PROCESSLIST` statement is very useful if you get a “too many connections” error message and want to determine which statements are executing. MySQL reserves one extra connection to be used by accounts that have the `SUPER` privilege. This ensures that even if the connection limit has been reached, an administrator can always connect and check the system, assuming that application users do not have the `SUPER` privilege.

Use the `KILL` statement to kill processes. If the process is running in a terminal that you can access, you can kill it with the `CTRL + C` keyboard combination, although this is less clean than using `KILL`.

# Disabling Client Access Control

To tell the server not to read the grant tables and disable access control, use the **--skip-grant-tables** option.

- Every connection succeeds:
  - You can provide any username and any password, and you can connect from any host.
  - The option disables the privilege system entirely.
  - Connected users effectively have all privileges.
- Prevent clients from connecting:
  - Use the **--skip-networking** option to prevent network access and allow access only on local socket, named pipe, or shared memory.
  - Use the **--socket** option to start the server on a non-standard socket to prevent casual access by local applications or users.

The **--skip-grant-tables** option has the following effects:

- When connected, the user has full privileges to do anything.
- This option disables account management statements such as `CREATE USER`, `GRANT`, `REVOKE`, and `SET PASSWORD`.

Disabling access control is convenient if you forget the `root` password and need to reset it, because any user can connect with full privileges without providing a password. This is clearly dangerous. To prevent remote clients from connecting over TCP/IP, use the **--skip-networking** option. Clients then can connect only from the `localhost` using a socket file on UNIX, or a named pipe or shared memory on Windows. To avoid casual connections from the local host, use a non-standard socket name at the command prompt.

Account management statements require the in-memory copies of the grant tables, which are not available when you disable access control. To change privileges or set a password, modify the grant tables directly. Alternatively, issue a `FLUSH PRIVILEGES` statement after connecting to the server, which causes the server to read the tables and also enables the account management statements.

# Setting Account Resource Limits

- Limit the use of server resources by setting the global **MAX\_USER\_CONNECTIONS** variable to a non-zero value.
  - This limits the number of simultaneous connections by any one account, but does not limit what a client can do when connected.
- Limit the following server resources for individual accounts:
  - **MAX\_QUERIES\_PER\_HOUR**: The number of queries that an account can issue per hour
  - **MAX\_UPDATES\_PER\_HOUR**: The number of updates that an account can issue per hour
  - **MAX\_CONNECTIONS\_PER\_HOUR**: The number of times an account can connect to the server per hour
  - **MAX\_USER\_CONNECTIONS**: The number of simultaneous connections allowed

To set resource limits for an account, use the `GRANT` statement with a `WITH` clause that names each resource to be limited. The default value for each limit is zero, indicating no limit. For example, to set limits for user `francis` to access the `customer` database, issue the following statement:

```
mysql> GRANT ALL ON customer.* TO 'francis'@'localhost'  
->      WITH MAX_QUERIES_PER_HOUR 20  
->      MAX_UPDATES_PER_HOUR 10  
->      MAX_CONNECTIONS_PER_HOUR 5  
->      MAX_USER_CONNECTIONS 2;
```

Provide resource limits in the `WITH` clause in any order. Set the `MAX_USER_CONNECTIONS` limit to 0 to set it to the global default, indicating that the maximum number of simultaneous connections allowed for this account is the global value of the `max_user_connections` system variable.

To reset an existing limit for any of the per-hour resources to the default of “no limit,” specify a value of 0, as in this example:

```
mysql> GRANT USAGE ON *.* TO 'quinn'@'localhost'  
->      WITH MAX_CONNECTIONS_PER_HOUR 0;
```

## Summary

In this lesson, you should have learned how to:

- Describe the user connection and query process
- List requirements for user authentication
- Create, modify, and drop user accounts
- Configure authentication plugins
- List requirements for user authorization
- Describe the levels of access privileges for users
- List the types of privileges
- Grant, modify, and revoke user privileges
- Use `SHOW PROCESSLIST` to display running threads
- Disable server client access control
- Set account resource limits

## Chapter 7

### Security

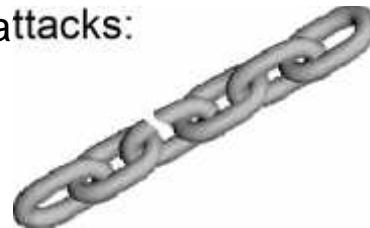
# **Objectives**

After completing this lesson, you should be able to:

- Recognize common security risks
- Describe security risks that are specific to the MySQL installation
- List security problems and counter-measures for networks, operating systems, file systems, and users
- Protect your data
- Use SSL for secure MySQL server connections
- Explain how SSH enables a secure remote connection to the MySQL server
- Find additional information about common security issues

# Security Risks

- MySQL server security is at risk when multiple users access it concurrently, particularly when such users connect over the Internet.
- It is not only the MySQL server that is at risk. The entire server host can be compromised
- There are many types of security attacks:
  - Eavesdropping
  - Altering
  - Playback
  - Denial of service
- MySQL uses security based on ACLs for all connections, queries, and other operations.
- ACLs also support SSL-encrypted connections between MySQL clients and servers.



You must keep data stored in MySQL databases secure to avoid exposing data that MySQL users expect to be private.

MySQL uses security based on access control lists (ACLs).

# MySQL Installation Security Risks

The most common installation security risks:

- Network
    - MySQL server permits clients to connect over the network and make requests.
    - Users can see client account information if privileges are not restricted.
    - Any accounts without passwords are vulnerable to attack.
  - Operating system
    - Extra user accounts on a server can be used to undermine the MySQL installation.
  - File system
    - Directories, database files, and log files on the server can be opened by users who should not have access.
- 
- **Network security:** Information about client accounts is stored in the `mysql` database. Set up each account with privileges that provide access only to the data that the account needs to see or modify. Assign account passwords to make it difficult to connect to the server using someone else's account. For example, a MySQL `root` account has full privileges to perform any database operation, so it is important to assign it a password that is not easily guessed.
  - **Operating system security:** You normally administer MySQL using a login account dedicated to that purpose. However, that account host might also have other login accounts. Minimizing the number of accounts that are not related to MySQL minimizes this risk.
  - **File system security:** The directories and files are part of the file system, so you need to protect them from direct access by other users who have login accounts on the server host. A MySQL installation also includes the programs and scripts used to manage and access databases. Users need to be able to run some of these (such as the client programs) but should not be able to modify or replace them.

Network security can be thought of as the highest level of defense. Within it is the operating system security. Inside that is the file system security, and even deeper is user security. Even though each level of security has its flaws (or holes), the levels form a virtually impenetrable fortress when they are combined.

# Network Security

Risk-prevention tasks:

- Invest in a firewall.
- Ensure access only to authorized clients.
- Limit network interfaces used by the server.
- Use a secure installation script:  
**`mysql_secure_installation`**
- Adhere to general privilege precautions.
- Do not transmit plain (unencrypted) data over the Internet.

The MySQL server operates in the client/server environment and provides an inherently network-oriented service. It is important to make sure that only authorized clients can connect to the server to access its databases. It is also important to make sure that MySQL accounts are protected with passwords and do not have unnecessary privileges. In addition, consider limiting the network interfaces used by the server.

There are many good, free open-source firewalls. MySQL is not intended to be facing the Internet. Special care should be taken if MySQL is not placed in the DMZ. In fact, if MySQL is running on the same machine as an Internet-facing application, you should probably use file sockets only.

Unencrypted data is accessible to any users who have the time and ability to intercept it and use it for their own purposes. Therefore, you should use an encrypted protocol such as SSL or SSH.

# Password Security

Risk-prevention tasks:

- Secure initial MySQL accounts with strong passwords.
- Do not store any plaintext passwords in your database.
  - The `mysql` database stores passwords in the `user` table.
  - It is better to store these passwords using one-way hashes.
- Do not choose passwords from dictionaries.
- Consider using the `validate_password` plugin to enforce a password policy.

If you use plaintext passwords and your computer becomes compromised, the intruder can take the full list of passwords and use them. Instead, use `MD5()`, `SHA1()`, `SHA2()`, or some other one-way hashing function and store the hash value. This prudent method is used to store other passwords in the server.

Special programs exist to break passwords. One method to counter these programs is to use a password that is taken from the first characters of each word in a sentence (for example, “Mary had a little lamb” results in a password of “Mhall”). The password is easy to remember and type, but difficult to guess for someone who does not know the sentence.

To enforce secure passwords, consider using the `validate_password` plugin, which is covered in the lesson titled “User Management”.

# Operating System Security

- Related to the complexity of the configuration:
  - Minimize the number of non-MySQL-related tasks on the server host.
- Additional usage introduces potential risk.
- Login accounts are not necessary for MySQL-only machines.
- There is a performance benefit to a system that is fully devoted to MySQL.

Minimize the number of server host tasks that do not directly relate to running MySQL. When you configure a host for fewer tasks, it can be made secure more easily than a host running a complex configuration that supports many services.

It is best to assign the MySQL server machine primarily or exclusively for MySQL, and not for other purposes such as web hosting or mail processing, or as a machine that hosts login accounts for general-purpose interactive use.

If other users can log in, there is a risk of exposing database information that should be kept private to the MySQL installation and its administrative account. For example, improper file system privileges can expose data files. Users can run the `ps` command to view information about processes and their execution environment.

When you use a machine only for MySQL, there is no need to have login accounts except the system administrative accounts and any other accounts that might be needed for administering MySQL itself (such as the account for the `mysql` user). Also, if you run fewer network services on the server host, fewer network ports need to be kept open. Closing ports minimizes the number of avenues of attack to which the host is exposed.

There is also a performance benefit to minimizing the number of non-MySQL services – more of your system's resources can be devoted to MySQL.

# File System Security

MySQL installation (directories and files) risk-prevention tasks:

- Change ownership and access permissions before starting the server.
  - Set multi-user system ownership to an account with administrative privileges.
  - Set MySQL-related directories and files and `user` and `group` table ownership to `mysql`, including:
    - MySQL programs
    - Database directories and files
    - Log, status, and configuration files
- Do not set passwords before protecting files. This can permit an unauthorized user to replace the files.
- Set up a dedicated account for MySQL administration.

Do not ever give anyone (except MySQL `root` accounts) access to the `user` table in the `mysql` database. Put MySQL behind the firewall or in a demilitarized zone (DMZ).

You can set passwords for the MySQL `root` account only while the server is running. Consequently, before starting the server and setting passwords, take any actions necessary to protect MySQL-related portions of the file system. If passwords are set first before protecting the files, it is possible for someone with direct file system access on the server host to replace the files. This compromises the MySQL installation and undoes the effect of setting the passwords.

For multi-user systems such as Linux, set ownership of all components of a MySQL installation to a dedicated login account with proper administrative privileges. This protects the installation against access by users who are not responsible for database administration. An additional benefit of setting up this account is that it can be used to run the MySQL server, rather than running the server from the Linux root account. A server that has the privileges of the `root` login account has more file system access than necessary and constitutes a security risk.

**Note:** This section assumes the existence of an administrative account that has `mysql` as both its username and its group name. However, the details of creating login accounts vary according to the version of Linux and are outside the scope of this course. Consult the documentation for your operating system.

# Protecting Your Data

There are many ways that users can corrupt data. You must take measures to protect it from attacks such as SQL injection.

- Do not trust any data entered by users of your applications.
  - Users can exploit application code by using characters with special meanings, such as quotes or escape sequences.
  - Make sure that your application remains secure if a user enters something like `DROP DATABASE mysql;`.
- Protect numeric and string data values.
  - Otherwise, users can gain access to secure data and submit queries that can destroy data or cause excessive server load.
- Protect even your publicly available data.
  - Attacks can waste server resources.
  - Safeguard web forms, URL names, special characters, and so on.

If your application generates a query such as `SELECT * FROM table WHERE ID=234` when a user enters the value 234, the user can enter the value `234 OR 1=1` to cause the application to generate the query `SELECT * FROM table WHERE ID=234 OR 1=1`. As a result, the server retrieves every row in the table. This exposes every row and causes excessive server load. To protect from this type of attack, use stored routines or prepared statements that do not interpret values as SQL expressions.

You might think that if a database contains only publicly available data, it need not be protected. This is incorrect. Even if it is permissible to display any row in the database, you should still protect against denial-of-service attacks or attempts to modify data by injecting `INSERT`, `UPDATE`, `REPLACE`, or `DELETE` statements. Otherwise, your data becomes unavailable to legitimate users. These are some techniques to detect this problem:

- Enter single and double quotation marks ('' and "") in all of your web forms.
- Modify dynamic URLs by adding %22 (""), %23 (#), and %27 ('') to them.
- Enter characters, spaces, and special symbols rather than numbers in numeric fields. Your application should remove them before passing them to MySQL; otherwise, an error is generated.

# Using Secure Connections

- MySQL uses unencrypted client-to-server connections by default.
  - Keeps standard configuration of MySQL as fast as possible
- Unencrypted connections are not acceptable for moving data securely over a network.
  - Network traffic is susceptible to monitoring and attack.
  - Data could be changed in transit between client and server.
- Encryption algorithms can be used to resist most threats.
  - They make any kind of data unreadable.
  - This resists many kinds of attacks.
- Some applications require the security that is provided by encrypted connections, wherein the extra computation is warranted.

If you use unencrypted connections between the client and the server, a user with access to the network can watch all the traffic and look at (and modify) the data being sent or received. Current industry practice requires many additional security elements from encryption algorithms. This resists many kinds of known attacks, such as changing the order of encrypted messages or replaying data twice.

Encrypting data is a CPU-intensive operation that requires the computer to do additional work and can delay other MySQL tasks. For applications that require the security provided by encrypted connections, the extra computation is warranted.

# SSL Protocol

MySQL supports SSL (secure sockets layer) connections between MySQL clients and the server.

- SSL is a connection protocol that:
  - Uses different encryption algorithms to secure data over a public network
  - Detects any data change, loss, or replay
  - Incorporates algorithms that provide identity verification using the X509 standard
- Applications requiring extra security (encryption) should use SSL.
- Secure connections are based on the OpenSSL API.

MySQL enables encryption on a per-connection basis. Depending on the requirements of individual applications, you can choose a normal unencrypted connection or a secure encrypted SSL connection.

Secure connections are based on the OpenSSL API and are available through the MySQL C API. Replication uses the C API, so secure connections can be used between master and slave servers.

X509 makes it possible to identify someone on the Internet. It is most commonly used in e-commerce applications. In basic terms, there should be a trusted Certificate Authority (CA) that assigns electronic certificates to anyone who needs them. Certificates rely on asymmetric encryption algorithms that have two encryption keys (a public key and a private key).

A certificate owner can provide the certificate to another party as proof of identity. A certificate contains its owner's public key and other details, signed by the trusted CA. Any data encrypted with this public key can be decrypted only by using the corresponding private key, which is held by the owner of the certificate.

For example, when you visit a secure (HTTPS) e-commerce site, the site provides your browser with its certificate. Your browser validates the certificate against its list of trusted CAs, and uses the public key it contains to create encrypted session information that can only be decrypted by the originating server. The browser and server can then communicate securely.

# Using SSL with the MySQL Server

- SSL usage requirements:
  - The system must support yaSSL (which comes bundled with MySQL) or OpenSSL.
  - The MySQL version being used must be built with SSL support.
- To get secure connections to work with MySQL and SSL, you must first do the following:
  - Load OpenSSL (if you are not using a precompiled MySQL).
  - Configure MySQL with SSL support.
  - Make sure that the `user` table in the `mysql` database includes the SSL-related columns (`ssl_*` and `x509_*`).
  - Check whether a server binary is compiled with SSL support, using the `--ssl` option.
- Start the server with options permitting clients to use SSL.

To make it easier to use secure connections, MySQL is bundled with yaSSL. (MySQL and yaSSL employ the same licensing model, whereas OpenSSL uses an Apache-style license.)

To obtain OpenSSL, go to <http://www.openssl.org>. Building MySQL using OpenSSL requires a shared OpenSSL library; otherwise, linker errors occur.

To configure a MySQL source distribution to use SSL, invoke CMake:

```
shell> cmake . -DWITH_SSL=bundled
```

- This configures the distribution to use the bundled yaSSL library.
- To use the system SSL library instead, specify the option as `-DWITH_SSL=system`.

If your `user` table does not have the SSL-related columns (beginning with `ssl_*` and `x509_*`), it must be upgraded using the `mysql_upgrade` program.

When checking whether a server binary is compiled with SSL support, an error occurs if the server does not support SSL:

```
shell> mysqld --ssl --help
060525 14:18:52 [ERROR] mysqld: unknown option '--ssl'
```

## Starting the MySQL Server with SSL

Enable the `mysqld` server clients to connect using SSL, with the following options:

- `--ssl-ca`: Identifies the Certificate Authority (CA) certificate to be used (required for encryption)
- `--ssl-key`: Identifies the server public key
- `--ssl-cert`: Identifies the server private key

```
shell> mysqld --ssl-ca=ca-cert.pem \
              --ssl-cert=server-cert.pem \
              --ssl-key=server-key.pem
```

`--ssl-cert` can be sent to the client and authenticated against the CA certificate that it is using.

# Requiring SSL-Encrypted Connections

- Use the **REQUIRE SSL** option of the **GRANT** statement to permit only SSL-encrypted connections for an account:

```
GRANT ALL PRIVILEGES ON test.*  
TO 'root'@'localhost'  
IDENTIFIED BY 'goodsecret' REQUIRE SSL;
```

- Start the **mysql** client with the **--ssl-ca** option to acquire an encrypted connection.
- (Optional) You can also specify the **--ssl-key** and **--ssl-cert** options for an X509 connection:

```
shell> mysql --ssl-ca=ca-cert.pem \  
--ssl-cert=server-cert.pem \  
--ssl-key=server-key.pem
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To establish a secure connection to a MySQL server with SSL support, the client options that you must specify depend on the SSL requirements of the user account that the client uses. MySQL can check X509 certificate attributes in addition to the usual authentication that is based on the username and password. To specify SSL-related options for a MySQL account, use the **REQUIRE** clause of the **GRANT** statement.

There are a number of ways to limit connection types for a given account. For the purposes of this course, only three options are described:

- **REQUIRE NONE**: Indicates that the account has no SSL or X509 requirements. This is the default if no SSL-related **REQUIRE** options are specified. Unencrypted connections are permitted if the username and password are valid. However, a client can also request an encrypted connection by specifying an option, if the client has the proper certificate and key files. That is, the client need not specify any SSL command options, in which case the connection is unencrypted.
- **REQUIRE SSL**: Tells the server to permit only SSL-encrypted connections for the account
- **REQUIRE X509**: The client must have a valid certificate, but the exact certificate, issuer, and subject do not matter. The only requirement is that it should be possible to verify its signature with one of the CA certificates.

## Checking SSL Status

- Check whether a running **mysqld** server supports SSL, using the value of the **have\_ssl** system variable:

```
mysql> SHOW VARIABLES LIKE 'have_ssl';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_ssl     | YES   |
+-----+
```

- Check whether the current server connection uses SSL, using the value of the **ssl\_cipher** status variable:

```
mysql> SHOW STATUS LIKE 'ssl_cipher';
+-----+-----+
| Variable_name | Value      |
+-----+-----+
| ssl_cipher    | DHE-RSA-AES256-SHA |
+-----+
```

If the value of `have_ssl` is YES, the server supports SSL connections. If the value is DISABLED, the server supports SSL connections but the appropriate `--ssl-*` options have not been provided at startup.

For the `mysql` client, you can use the `STATUS` or `\s` command and check the SSL line:

```
mysql> \s
...
SSL:                               Cipher in use is DHE-RSA-AES256-SHA
...
```

## Advantages and Disadvantages of Using SSL

| Advantages                               | Disadvantages                     |
|--|-----------------------------------|
| Added security for applications in need  | CPU-intensive                     |
| Can be enabled on a per-connection basis | Slows down client/server protocol |
| Can be used with replication             | Can delay other SQL tasks         |

## Secure Remote Connection to MySQL

- MySQL supports SSH (secure shell) connection to a remote MySQL server. This requires:
  - An SSH client on the client machine
  - Port forwarding through an SSH tunnel from the client to the server
  - A client application on the machine with the SSH client
- When you complete the setup, you have a local port that hosts an SSH connection to MySQL, encrypted using SSH.

## **Summary**

In this lesson, you should have learned how to:

- Recognize common security risks
- Describe security risks that are specific to the MySQL installation
- List security problems and counter-measures for networks, operating systems, file systems, and users
- Protect your data
- Use SSL for secure MySQL server connections
- Explain how SSH enables a secure remote connection to the MySQL server
- Find additional information about common security issues

## **Chapter 8**

Table Maintenance and Exporting-Importing

## **Objectives**

After completing this lesson, you should be able to:

- Recognize the types of table maintenance operations
- Execute SQL statements for table maintenance
- Use client and utility programs for table maintenance
- Maintain tables according to their specific storage engine

## Implementation of Table Maintenance

- Table maintenance operations are useful for identifying and correcting database problems such as the following:
  - Tables that become damaged as a result of a server crash
  - Slow query processing on tables
- Many tools are available for performing table maintenance:
  - MySQL Workbench
  - MySQL Enterprise Monitor
  - SQL (DML) maintenance statements
  - Utilities:
    - `mysqlcheck`
    - `myisamchk`
  - Server auto-recovery

# SQL for Table Maintenance Operations

- There are multiple SQL statements for performing table maintenance:
  - **ANALYZE TABLE**: Updates index statistics
  - **CHECK TABLE**: Checks the integrity of the table
  - **CHECKSUM TABLE**: Reports a checksum for the table
  - **REPAIR TABLE**: Repairs the table
  - **OPTIMIZE TABLE**: Optimizes the table
- Each statement takes one or more table names and optional keywords.
- Example of a maintenance statement and output:

| mysql> CHECK TABLE world.innodb.City; |       |          |          |
|---------------------------------------|-------|----------|----------|
| Table                                 | Op    | Msg_type | Msg_text |
| world.innodb.City                     | check | status   | OK       |

After performing the requested operation, the server returns information about the result of the operation to the client. The information takes the form of a result set with four columns:

- **Table**: Indicates the table for which the operation was performed
- **Op**: Names the operation (check, repair, analyze, or optimize)
- **Msg\_type**: Provides an indicator of success or failure
- **Msg\_text**: Provides extra information

## ANALYZE TABLE Statement

- Analyzes and stores the key distribution statistics of a table
- Is used to make better choices about query execution
- Works with InnoDB, NDB, and MyISAM tables
- Supports partitioned tables
- **ANALYZE TABLE** option:
  - **NO\_WRITE\_TO\_BINLOG** or **LOCAL**: Suppress binary log
- Example of a good **ANALYZE TABLE** result:

| mysql> ANALYZE LOCAL TABLE Country; |         |          |          |
|-------------------------------------|---------|----------|----------|
| Table                               | Op      | Msg_type | Msg_text |
| world.innodb.Country                | analyze | status   | OK       |

MySQL uses the stored key distribution statistics to decide the order in which the optimizer joins tables when you perform a join on something other than a constant. In addition, key distributions determine which indexes MySQL uses for a specific table within a query.

You can execute the **ANALYZE TABLE** statement to analyze and store the statistics, or you can configure InnoDB to gather the statistics automatically after a number of data changes or when you query table or index metadata.

**ANALYZE TABLE** characteristics:

- During the analysis, MySQL locks the table with a read lock for InnoDB and MyISAM.
- This statement is equivalent to using `mysqlcheck --analyze`.
- Requires **SELECT** and **INSERT** privileges for the table
- Supports partitioned tables. You can also use `ALTER TABLE...ANALYZE PARTITION` to check one or more partitions.

If the table has not changed since running the last **ANALYZE TABLE** statement, MySQL does not analyze the table.

By default, MySQL writes **ANALYZE TABLE** statements to the binary log and replicates them to replication slaves. Suppress logging with the optional **NO\_WRITE\_TO\_BINLOG** keyword or its alias, **LOCAL**.

You can control how MySQL gathers and stores key distribution statistics with the following options:

- **innodb\_stats\_persistent**: When this option is ON, MySQL enables the STATS\_PERSISTENT setting on newly created tables. You can also set STATS\_PERSISTENT on tables when using the CREATE TABLE or ALTER TABLE statements. By default, MySQL does not persist key distribution statistics to disk, so they must be generated at times such as server restart. MySQL stores key distribution statistics on disk for tables with STATS\_PERSISTENT enabled, so that it does not need to generate statistics as frequently for those tables. This allows the optimizer to create more consistent query plans over time.
- **innodb\_stats\_persistent\_sample\_pages**: MySQL recalculates statistics by reading a sample of a STATS\_PERSISTENT table's index pages, rather than the entire table. By default, it reads a sample of 20 pages. Increase this number to improve the quality of generated statistics and query plans. Decrease this number to reduce the I/O cost of generating statistics.
- **innodb\_stats\_transient\_sample\_pages**: This option controls the number of index pages sampled on tables that do not have the STATS\_PERSISTENT setting.

The following options control how MySQL gathers statistics automatically.

- **innodb\_stats\_auto\_recalc**: With this option enabled, MySQL generates statistics automatically for a STATS\_PERSISTENT table when 10% of its rows have changed since the last recalculation.
- **innodb\_stats\_on\_metadata**: Enable this option to update statistics when you execute metadata statements such as SHOW TABLE STATUS, or when querying INFORMATION\_SCHEMA.TABLES. By default, this option is disabled.

## CHECK TABLE Statement

- Checks the integrity of table structure and contents for errors
- Verifies view definitions
- Supports partitioned tables
- Works with InnoDB, CSV, MyISAM, and ARCHIVE tables
- **CHECK TABLE** options:
  - **FOR UPGRADE**: Check whether tables work with the current server.
  - **QUICK**: Do not scan rows for incorrect links.
- If **CHECK TABLE** finds a problem for an InnoDB table:
  - The server shuts down to prevent error propagation
  - MySQL writes an error to the error log

**CHECK TABLE** characteristics:

- For MyISAM tables, the key statistics are updated as well.
- Can also check views for problems, such as tables that are referenced in the view definition that no longer exist
- Supports partitioned tables. You can also use **ALTER TABLE...CHECK PARTITION** to check one or more partitions.

With **FOR UPGRADE**, the server checks each table to determine whether the table structure is compatible with the current version of MySQL. Incompatibilities might occur because the storage format for a data type has changed or because its sort order has changed. If there is a possible incompatibility, the server runs a full check on the table. If the full check succeeds, the server marks the table's **.frm** file with the current MySQL version number. Marking the **.frm** file ensures that future checks for the table with the same version of the server are fast.

Use **FOR UPGRADE** with the InnoDB, MyISAM, and ARCHIVE storage engines.

Use **QUICK** with InnoDB and MyISAM tables. MyISAM supports other options. See <http://dev.mysql.com/doc/mysql/en/check-table.html> for more information.

## CHECK TABLE Statement

Example of a good **CHECK TABLE** result:

| mysql> CHECK TABLE Country; |       |          |          |
|-----------------------------|-------|----------|----------|
| Table                       | Op    | Msg_type | Msg_text |
| world_innodb.Country        | check | status   | OK       |

If the output from **CHECK TABLE** indicates that a table has problems, repair the table. For example, you could use the **CHECK TABLE** statement to detect hardware problems (such as faulty memory or bad disk sectors) before repairing a table. The **Msg\_text** output column is normally **OK**. If you do not get either **OK** or **Table** is already up to date, run a repair of the table.

If the table is marked as **corrupted** or not **closed** properly but **CHECK TABLE** does not find any problems in the table, it marks the table as **OK**.

## CHECKSUM TABLE Statement

- Reports a table **checksum**
  - Is used to verify that the contents of a table are the same before and after a backup, rollback, or other operation
- Reads the entire table row by row to calculate the checksum
  - The default **EXTENDED** option provides this behavior.
  - A **QUICK** option is available on MyISAM tables.
    - This is the default option with the MyISAM **CHECKSUM=1** setting.
- Example of a **CHECKSUM TABLE** statement:

```
mysql> CHECKSUM TABLE City;
+-----+-----+
| Table | Checksum |
+-----+-----+
| world_innodb.City | 531416258 |
+-----+-----+
```

### CHECKSUM TABLE characteristics:

- **CHECKSUM TABLE** requires the **SELECT** privilege for the table.
- For a nonexistent table, **CHECKSUM TABLE** returns **NULL** and generates a warning.
- If the **EXTENDED** option is used, the entire table is read row by row, and the **checksum** is calculated.
- If the **QUICK** option is used:
  - A live table **checksum** is reported if it is available; **NULL** is reported otherwise. This is very fast.
  - Enable live **checksum** on MyISAM tables by specifying the **CHECKSUM=1** table option when you create the table.
- If neither **QUICK** nor **EXTENDED** is specified, MySQL assumes **EXTENDED** except for MyISAM tables with **CHECKSUM=1**.

The **checksum** value depends on the table row format. If the row format changes, the **checksum** also changes. For example, the storage format for **VARCHAR** changed after MySQL 4.1, so if you upgrade a 4.1 table to a later version, the **checksum** value changes if the table contains **VARCHAR** fields.

**Note:** If the checksums for two tables are different, it is likely that the tables differ in some way. However, since the hashing function used by CHECKSUM TABLE is not guaranteed to be collision-free, there is a very remote possibility that two non-identical tables can produce the same checksum.

## OPTIMIZE TABLE Statement

- Cleans up a table by defragmenting it
  - Defragments by rebuilding the table and freeing unused space
- Locks the table during optimization
- Updates index statistics
- Is most beneficial on a permanent, fully populated table
- Works with InnoDB, MyISAM, and ARCHIVE tables
- Supports partitioned tables
- **OPTIMIZE TABLE** option:
  - **NO\_WRITE\_TO\_BINLOG** or **LOCAL**: Suppress the binary log.

OPTIMIZE TABLE characteristics:

- Defragmenting involves reclaiming unused space caused by deletes and updates, and coalescing records that have become split and stored non-contiguously.
- Requires `SELECT` and `INSERT` privileges for the table
- Supports partitioned tables. You can also use `ALTER TABLE...OPTIMIZE PARTITION` to check one or more partitions.

For example, you can use the `OPTIMIZE TABLE` statement to rebuild a `FULLTEXT` index in InnoDB, after modifying a substantial number of rows.

With InnoDB tables, `OPTIMIZE TABLE` is mapped to `ALTER TABLE`, which rebuilds the table to update index statistics and free unused space in the clustered index. InnoDB is not subject to fragmentation in the same way as other storage engines, so you do not need to use `OPTIMIZE TABLE` very often.

Use `OPTIMIZE TABLE` on a table that uses the ARCHIVE storage engine to compress the table. The number of rows in ARCHIVE tables reported by `SHOW TABLE STATUS` is always accurate. An `.ARN` file may appear during optimization operations.

## OPTIMIZE TABLE Statement

The following **OPTIMIZE TABLE** statement optimizes two fully populated tables in the mysql database:

```
mysql> OPTIMIZE TABLE mysql.help_relation, mysql.help_topic;
+-----+-----+-----+-----+
| Table | Op   | Msg_type | Msg_text |
+-----+-----+-----+-----+
| mysql.help_relation | optimize | status   | OK
| mysql.help_topic   | optimize | status   | OK
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

With MyISAM tables, use the `OPTIMIZE TABLE` statement after a large part of a table has been deleted or if many changes have been made to a table with variable-length rows (tables that have `VARCHAR`, `VARBINARY`, `BLOB`, or `TEXT` columns). Deleted rows are maintained in a linked list, and subsequent `INSERT` operations reuse old row positions. `OPTIMIZE TABLE` works best when you use it on a table that is fully populated and does not change much. The benefits of optimization diminish if the data changes a lot, and you have to optimize often.

## REPAIR TABLE Statement

- Repairs a possibly corrupted MyISAM or ARCHIVE table
  - Does not support InnoDB
- Supports partitioned tables
- **REPAIR TABLE** options:
  - **QUICK**: Repair only the index tree.
  - **EXTENDED**: Create index row by row (instead of creating one index at a time with sorting).
  - **USE\_FRM**: Re-create the **.MYI** file by using the **.FRM** file.
  - **NO\_WRITE\_TO\_BINLOG** or **LOCAL**: Suppress the binary log.

REPAIR TABLE characteristics:

- **QUICK option**: Tries to repair only the index file and not the data file. This type of repair is like that done by `myisamchk --recover --quick`.
- **EXTENDED option**: MySQL creates the index row by row instead of creating one index at a time with sorting. This type of repair is like that done by `myisamchk --safe-recover`.
- The **USE\_FRM** option cannot be used on a partitioned table.
- Requires **SELECT** and **INSERT** privileges for the table
- Supports partitioned tables. You can also use **ALTER TABLE...REPAIR PARTITION** to check one or more partitions.

It is best to make a backup of a table before performing a table repair operation; under some circumstances, the operation might cause data loss. Possible causes include (but are not limited to) file system errors.

If the server crashes during a **REPAIR TABLE** operation, you should immediately execute another **REPAIR TABLE** after restarting before performing any other operations to avoid further corruption.

If you frequently need to use **REPAIR TABLE** to recover from corrupt tables, try to find the underlying reason to prevent such corruption and eliminate the need to use **REPAIR TABLE**.

## REPAIR TABLE Statement

Example of a **REPAIR TABLE** statement:

| Table                   | Op     | Msg_type | Msg_text |
|-------------------------|--------|----------|----------|
| mysql.help_relation     | repair | status   | OK       |
| 1 row in set (0.00 sec) |        |          |          |

## **mysqlcheck Client Program**

- Is a command-line client that checks, repairs, analyzes, and optimizes tables
- Can be more convenient than issuing SQL statements
- Works with InnoDB, MyISAM, and ARCHIVE tables
- Three levels of checking:
  - Table-specific
  - Database-specific
  - All databases
- Some **mysqlcheck** maintenance options:
  - **--analyze**: Perform an **ANALYZE TABLE**.
  - **--check**: Perform a **CHECK TABLE** (default).
  - **--optimize**: Perform an **OPTIMIZE TABLE**.
  - **--repair**: Perform a **REPAIR TABLE**.

In some cases, `mysqlcheck` is more convenient than issuing SQL statements directly. For example, if you provide it with a database name as its argument, `mysqlcheck` determines what tables the database contains and issues statements to process them all. You do not need to provide explicit table names as arguments. Also, because `mysqlcheck` is a command-line program, you can easily use it in operating system jobs that perform scheduled maintenance.

## mysqlcheck Client Program

- Oracle recommends that you run **mysqlcheck** with no options first. Run it again if repairs are needed.
- Some **mysqlcheck** modification options:
  - **--repair --quick**: Attempt a quick repair.
  - **--repair**: Repair normally (if quick repair fails).
  - **--repair --force**: Force a repair.
- **mysqlcheck** examples:

```
shell> mysqlcheck --login-path=admin world_innodb City Country
shell> mysqlcheck --login-path=admin \
--databases world_innodb test
shell> mysqlcheck -uroot -p mysql user --repair
shell> mysqlcheck -uroot -p --all-databases
shell> mysqlcheck --login-path=admin --analyze --all-databases
```

By default, `mysqlcheck` interprets its first non-option argument as a database name and checks all the tables in that database. If any other arguments follow the database name, it treats them as table names and checks just those tables.

The `mysqlcheck` examples shown in the slide demonstrate the following:

- The first of the commands (with options) checks only the `City` and `Country` tables in the `world_innodb` database.
- In the example with the `--databases` (or `-B`) option, `mysqlcheck` interprets its arguments as database names and checks the tables in the `world_innodb` and `test` databases.
- With the `--all-databases` (or `-A`) option, `mysqlcheck` checks all tables in all databases.

## **myisamchk Utility**

- Is a non-client utility that checks MyISAM tables
- Is similar to **mysqlcheck** with some of the following differences:
  - It can enable or disable indexes.
  - It accesses table files directly rather than through the server.
  - This avoids concurrent table access.
- Some **myisamchk** options:
  - **--recover**: Repair a table.
  - **--safe-recover**: Repair a table that **--recover** cannot repair.
- **myisamchk** examples:

```
shell> myisamchk /var/lib/mysql/mysql/help_topic
shell> myisamchk help_category.MYI
shell> myisamchk --recover help_keyword
```

Conceptually, **myisamchk** is similar in purpose to **mysqlcheck**. However, **myisamchk** does not communicate with the MySQL server. Instead, it accesses the table files directly.

Perform table maintenance while avoiding concurrent table access with **myisamchk**:

1. Ensure that the server does not access the tables while they are being worked on. One way to guarantee this is to lock the tables or stop the server.
2. From a command prompt, change location into the database directory where the tables are located. This is the subdirectory of the server's data directory that has the same name as the database containing the tables to be checked. (The reason for changing location is to make it easier to refer to the table files. This step can be skipped, but **myisamchk** has to include the directory where the tables are located.)
3. Invoke **myisamchk** with options indicating the operation to be performed, followed by arguments that name the tables on which **myisamchk** should operate. Each of these arguments can be either a table name or the name of the table's index file. An index file name is the same as the table name, plus an **.MYI** suffix. Therefore, a table can be referred to either as *table\_name* or as *table\_name.MYI*.
4. Restart the server.

**Note:** Try **--recover** first, because **--safe-recover** is much slower.

## Options for mysqlcheck and myisamchk

Options to control the type of maintenance performed:

| Both  | mysqlcheck only                                     | myisamchk only                          |
|---|---|---|
| --analyze or -a<br>--check or -c<br>--check-only-changed or -C<br>--fast or -F<br>--medium-check or -m<br>--quick or -q | --auto-repair<br>--extended or -e<br>--repair or -r | --extend-check or -e<br>--recover or -r |

mysqlcheck and myisamchk both take many options to control the type of table maintenance operation performed. The table in the slide summarizes some of the more commonly used options, most of which are understood by both programs. Where that is not the case, it is noted in the relevant option description.

- **--analyze:** Analyze the distribution of key values in the table. This can improve performance of queries by speeding up index-based lookups.
- **--auto-repair:** Repair tables automatically if a check operation discovers problems.
- **--check or -c:** Check tables for problems. This is the default action if no other operation is specified.
- **--check-only-changed or -C:** Skip table checking except for tables that have been changed since they were last checked or tables that have not been properly closed. The latter condition might occur if the server crashes while a table is open.
- **--fast or -F:** Skip table checking except for tables that have not been properly closed.
- **--extended, --extend-check, or -e:** Run an extended table check. For mysqlcheck, when this option is given in conjunction with a repair option, a more thorough repair is performed than when the repair option is given alone. That is, the repair operation performed by --repair --extended is more thorough than the operation performed by --repair.

- **--medium-check** or **-m**: Run a medium table check.
- **--quick** or **-q**: For `mysqlcheck`, **--quick** without a repair option causes only the index file to be checked, leaving the data file alone. For both programs, **--quick** in conjunction with a repair option causes the program to repair only the index file, leaving the data file alone.
- **--repair**, **--recover**, or **-r**: Run a table repair operation.

# InnoDB Table Maintenance

- InnoDB recovers automatically after a failure.
- Use CHECK TABLE or a client program to find inconsistencies, incompatibilities, and other problems.
- Restore a table by dumping it with **mysqldump**:

```
shell> mysqldump <db_name> <table_name> > <dump_file>
```

- Then drop it and re-create it from the dump file.

```
shell> mysql <db_name> < <dump_file>
```

- To repair tables after a crash, restart the server with the **--innodb\_force\_recovery** option, or restore tables from a backup.
- Optimizing using ALTER TABLE rebuilds tables and frees unused space in clustered indexes.

If a table check indicates problems, restore the table to a consistent state by dumping it with mysqldump, dropping it, and re-creating it from the dump file.

In the event of a crash of the MySQL server or the host on which it runs, some InnoDB tables might be in an inconsistent state. InnoDB performs auto-recovery as part of its startup sequence. Rarely, the server does not start due to failure of auto-recovery. If that happens, use the following procedure:

- Restart the server with the **--innodb\_force\_recovery** option set to a value from 1 to 6. These values indicate increasing levels of caution in avoiding a crash, and increasing levels of tolerance for possible inconsistency in the recovered tables. A good value to start with is 4, which prevents insert buffer merge operations.
- When the server is started with **--innodb\_force\_recovery** set to a non-zero value, InnoDB prevents INSERT, UPDATE, or DELETE operations. Therefore, you should dump the InnoDB tables and then drop them while the option is in effect. Then restart the server without the **--innodb\_force\_recovery** option. When the server comes up, recover the InnoDB tables from the dump files.
- If the preceding steps fail, restore the tables from a previous backup.

# MyISAM Table Maintenance

- Use CHECK TABLE . . . MEDIUM (the default option).
  - Use the client program to run **myisamchk**:

```
shell> myisamchk --medium-check <table_name>
```
  - Table corruption is usually in the indexes, making these checks very helpful.
  - The utility repairs the table if it is corrupted.
- Set up a server to run checks and repair tables automatically.
  - Enable auto-repair by using the **--myisam-recover** option.
  - The server checks each MyISAM table the first time it is accessed after startup to make sure that it was closed properly the last time.
- Force recovery of MyISAM tables from the config file.

The default CHECK TABLE check type is MEDIUM for both dynamic and static format tables. If a static-format table type is set to CHANGED or FAST, the default is QUICK. The row scan is skipped for CHANGED and FAST because the rows are very seldom corrupted.

CHECK TABLE changes the table if the table is marked as “corrupted” or “not closed properly.” If it does not find any problems in the table, it marks the table status as “up to date.” If a table is corrupted, the problem is most likely in the indexes and not in the data.

The **--myisam-recover** option value can consist of a comma-separated list of one or more of the following values:

- **DEFAULT**: Default checking
- **BACKUP**: Tells the server to make a backup of any table that it must change
- **FORCE**: Performs table recovery even when causing the loss of more than one row of data
- **QUICK**: Performs a quick recovery. The recovery skips tables that have no gaps (known as “holes”) between rows resulting from deletes or updates.

For example, to tell the server to perform a forced recovery of MyISAM tables found to have problems but to make a backup of any table that it changes, add the following to your option file:

```
[mysqld]
myisam-recover=FORCE, BACKUP
```

## MEMORY Table Maintenance

- MEMORY tables do not release memory when rows are deleted using a **DELETE...WHERE** statement.
- To release memory, you must perform a null **ALTER TABLE** operation.
- To see the storage requirements for a MEMORY table, execute **SHOW TABLE STATUS** and add the **Data\_length** and **Index\_length** fields.
  - The **Data\_free** field is nonzero if deleted rows have not been reused.

The following is the output of a **SHOW TABLE STATUS** statement on a copy of the `world_innodb` database `City` table called `CityCopy`. The `CityCopy` table uses the MEMORY storage engine:

```
mysql> SHOW TABLE STATUS LIKE 'CityCopy'\G
***** 1. row ****
      Name: CityCopy
      Engine: MEMORY
     Version: 10
   Row_format: Fixed
         Rows: 4079
 Avg_row_length: 67
  Data_length: 383072
Max_data_length: 10808373
  Index_length: 253984
  Data_free: 0
 ...
1 row in set (0.00 sec)
```

The total memory the `CityCopy` table occupies can be calculated as follows:  
`Data_length (383072) + Index_length (253984) = 637056 bytes`

## ARCHIVE Table Maintenance

- ARCHIVE compression issues:
  - Table rows are compressed as they are inserted.
  - On retrieval, rows are uncompressed on demand.
  - Some **SELECT** statements can deteriorate the compression.
- Use of **OPTIMIZE TABLE** or **REPAIR TABLE** can achieve better compression.
- **OPTIMIZE TABLE** works when the table is not being accessed (by read or write).

# Exporting and Importing Data

- Types of export/import operations available:
  - `SELECT...INTO OUTFILE`
  - `LOAD DATA INFILE`
- Uses for exporting data:
  - Copying databases from one server to another:
    - From within the same host
    - To another host
  - Testing a new MySQL release with real data
  - Transferring data from one RDBMS to another

The two most common ways to accomplish export and import operations are:

- Using `SELECT ... INTO OUTFILE` to export data to files
- Using the `LOAD DATA INFILE` statement to import data from files

## Reasons to Export Data

Database exports are useful for copying your databases to another server. You can transfer a database to a server running on another host, which is the most typical task when exporting data. You can also transfer data to a different server running on the same host. You might do this if you are testing a server for a new release of MySQL and want to use it with some real data from your production server. You can also load data into external applications; a data export is also useful for transferring data from one RDBMS to another.

## Exporting Data by Using SELECT with INTO OUTFILE

- **SELECT with INTO OUTFILE:**

- Writes result set directly into a file
- Assumes the file path is in the database data directory, unless otherwise specified

```
SELECT * INTO OUTFILE
  '/tmp/City.txt' FROM City;
```

- The output file:

- Is written to server host, instead of over the network to client
- Is created with file system access permissions
- Contains one line per row selected by the statement

You can use a SELECT statement with the INTO OUTFILE clause to write the results set directly into a file. To use SELECT in this way, place the INTO OUTFILE clause before the FROM clause.

The name of the file indicates the location of the output file. MySQL writes the file to the specified path on the server host.

### File Characteristics

The output file has the following characteristics:

- The file is written to the server host, instead of sending the file over the network to the client. The file must not already exist.
- The server writes a new file on the server host. To run the SELECT ... INTO OUTFILE statement, you must connect to the server by using an account that has the FILE privilege.
- MySQL creates the file with permissions as follows:
  - The account running the MySQL process owns the file.
  - The file is world-readable.
- The file contains one line per row selected by the statement. By default, column values are delimited by tab characters and lines are terminated with newlines.

# Using Data File Format Specifiers

- **SELECT** with **INTO OUTFILE** outputs TSV by default.
  - “Tab-separated values” files delimit columns with tab characters, and delimit records with newline characters.
- You can choose alternative delimiters:

| <b>FIELDS</b>                       |
|-------------------------------------|
| <b>TERMINATED BY</b> 'string'       |
| <b>ENCLOSED BY</b> 'char'           |
| <b>ESCAPED BY</b> 'char'            |
| <b>LINES TERMINATED BY</b> 'string' |

- The **FIELDS** clause specifies the characteristics of columns.
- **TERMINATED BY**, **ENCLOSED BY**, and **ESCAPED BY** use defaults if you do not specify a value.
- **LINES TERMINATED BY** specifies the row delimiter.

`SELECT...INTO OUTFILE` assumes a default data file format in which column values are separated by tab characters and records are terminated by newlines. To use `SELECT...INTO OUTFILE` to write a file with different separators or terminators, specify the format output format with **FIELDS** and **LINES** clauses.

- The **FIELDS** clause specifies how columns are represented.
  - **TERMINATED BY** specifies the field delimiter and is the tab character by default.
  - **ENCLOSED BY** specifies how column values are quoted. The default setting uses no quotes (that is, the default value is the empty string).
  - **ESCAPED BY** indicates the escape character used when indicating non-printing characters such as a newline or tab character. The default escape character is the backslash (\) character.
- The **LINES TERMINATED BY** clause specifies the row delimiter, which is a newline by default.

MySQL uses a backslash character to escape special characters, so you must represent characters such as newline and tab as '`\n`' and '`\t`', respectively. Similarly, to represent a literal backslash, you must escape it as follows: '`\\\`'.

# Escape Sequences

- Line terminator specifiers:
  - Newline character is the default.
  - Examples:
    - `LINES TERMINATED BY '\n'`
    - `LINES TERMINATED BY '\r'`
- **ESCAPED BY** option:
  - Handles only values in the data file
  - Can use '@'

| Sequence | Meaning            |
|----------|--------------------|
| \N       | NULL               |
| \0       | NULL (zero) byte   |
| \b       | Backspace          |
| \n       | Newline (linefeed) |
| \r       | Carriage return    |
| \s       | Space              |
| \t       | Tab                |
| \'       | Single quote       |
| \"       | Double quote       |
| \\       | Backslash          |

You can use all of the sequences shown in the slide alone or within a longer string, with the exception of \N, which is understood as NULL only when it appears alone.

Common line terminators include the newline and the carriage return/newline pair. The default newline terminators are common on Linux systems, and carriage return/newline pairs are common on Windows systems.

## ESCAPED BY

The ESCAPED BY clause controls only the output of values in the data file; it does not change how MySQL interprets special characters in the statement. For example, if you specify a data file escape character of '@' by writing `ESCAPED BY '@'`, that does not mean you must use '@' to escape special characters elsewhere in the statement. You must use MySQL's escape character (the backslash: \) to escape special characters in the statement by using syntax such as `LINES TERMINATED BY '\r\n'` rather than `LINES TERMINATED BY '@r@n'`.

# Importing Data by Using LOAD DATA INFILE

## LOAD DATA INFILE:

- Reads rows values from a file into a table
- Is the reverse operation of **SELECT... INTO OUTFILE**
- Uses similar clauses and format specifiers:
  - Example:

```
LOAD DATA INFILE '/tmp/City.txt'  
INTO TABLE City  
FIELDS TERMINATED BY ',';
```

- Assumes the file is located on the server host in the database data directory

The LOAD DATA INFILE statement reads the values from a data file into a table. LOAD DATA INFILE is the reverse operation of SELECT ... INTO OUTFILE.

## Importing Tab-Delimited or Comma-Separated Files

To import a data file containing tab-delimited or comma-separated table data, use the LOAD DATA INFILE command. The most important characteristics of the file are:

- The column value separators
- The line separator
- The characters that enclose the values (for example, quotation marks)
- Whether the column names are specified in the file
- Whether there is a header indicating rows of the table to skip before importing
- The location of the file
- Whether privileges are required to access the file
- The order of the columns
- Whether the number of columns in the file and the table match

## Importing Data from the Client Host

- By default, `LOAD DATA INFILE` reads the import file from the server's file system.
- Use `LOAD DATA LOCAL INFILE` to access files on the client host.
- Consider the following factors when using `LOAD DATA LOCAL INFILE`:
  - The import is slower because the file is sent from the client to the server for processing.
  - Data and duplicate key errors are downgraded to warnings and processing continues.
  - There is a potential security risk.

Importing data from the client host is slower than importing from the server's file system, because the import file is first uploaded to the server. The server creates a copy of the file in its temporary directory. Once transmission has started the server cannot stop it, so data interpretation and duplicate key errors that usually terminate the import are flagged as warnings instead and processing continues.

You need to take steps to avoid security risks by ensuring the server cannot read privileged files from the client. The `LOAD DATA INFILE` statement allows the server process to copy and read the contents of the file named in the statement. A malicious administrator could create a modified version of the server code that could read any file from the client, not just the file specified in the statement.

You can disable all `LOAD DATA LOCAL` statements from the server side by starting `mysqld` with the `--local-infile=0` option. Limit file import and export operations to a specific directory by adding the directory path to the `secure_file_priv` system variable.

For more information about managing security risks with `LOAD DATA LOCAL`, see the *MySQL Reference Manual*: <http://dev.mysql.com/doc/mysql/en/load-data-local.html>.

# Skipping or Transforming Input Data

- To ignore lines in the data file, use **IGNORE n LINES**:

```
mysql> LOAD DATA INFILE '/tmp/City.txt'  
-> INTO TABLE City IGNORE 2 LINES;  
  
Query OK, 2231 rows affected (0.01 sec)
```

- There are 2233 rows in the file, but only 2231 are loaded.
- To ignore or transform column values:
  - Specify user variables (instead of a column name) in the statement's column list.
  - (Optional) Transform columns by using a **SET** clause:

```
LOAD DATA INFILE '/tmp/City.txt' INTO  
TABLE City ( @skip, @Name,  
CountryCode, @District, Population)  
SET name=CONCAT(@Name, ' ',@District);
```

- The statement ignores the values of variables that are not used in a **SET** expression.

## Ignoring Data File Lines

To ignore the initial part of the data file, you can use the **IGNORE n LINES** clause, where *n* is an integer that indicates the number of input lines to ignore. Use this clause when a file begins with a row of column names rather than data values.

## Ignoring or Transforming Column Values

You can provide user variables in the column list and in the optional **SET** clause, which has a syntax similar to the **SET** clause of the **UPDATE** statement. **LOAD DATA INFILE** transforms data values that it reads from the file by processing the values contained in the user variables before inserting them into the table.

To assign an input data column to a user variable rather than to a table column, provide the name of a user variable in the column list. If the column is assigned to a user variable that is not used in a **SET** expression, the statement ignores the value in that column and does not insert it into the table.

# Duplicate Records

- To control how **LOAD DATA INFILE** handles rows that contain a duplicate primary or unique key:
  - Use the **IGNORE** keyword to discard rows with duplicate keys.
  - Use the **REPLACE** keyword to replace rows with the version from the file with the same key.
- This is similar to how you can control duplicates with the **INSERT** and **REPLACE** statements:
  - **IGNORE**
  - **ON DUPLICATE KEY UPDATE**

When you add new rows to a table by using the **INSERT** or **REPLACE** statements, you can control how the statement handles rows that contain duplicate keys already present in the table. You can allow the statement to raise an error, you can use the **IGNORE** clause to discard the row, or you can use the **ON DUPLICATE KEY UPDATE** clause to modify the existing row.

**LOAD DATA INFILE** provides the same degree of control over duplicate rows through the use of two modifier keywords, **IGNORE** and **REPLACE**.

However, its duplicate-handling behavior differs slightly depending on whether the data file is on the server host or on the client host, so you must consider the data file location when using **LOAD DATA INFILE**.

## Loading a File from the Server Host

When loading a file that is located on the server host, **LOAD DATA INFILE** handles rows that contain duplicate unique keys as follows:

- By default, an input record that causes a duplicate-key violation results in an error; the rest of the data file is not loaded. Records processed up to that point are loaded into the table.
- If you provide the **IGNORE** keyword after the file name, new records that cause duplicate-key violations are ignored, and the statement does not raise an error. **LOAD DATA INFILE** processes the entire file, loads all records not containing duplicate keys, and discards the rest.
- If you provide the **REPLACE** keyword after the file name, new records that cause duplicate-key violations replace any records already in the table that contain the duplicated key values. **LOAD DATA INFILE** processes the entire file and loads all its records into the table.

## Loading a File from the Client Host

When you load a file from the client host, **LOAD DATA INFILE** ignores records that contain duplicate keys by default. That is, the default behavior is the same as if you specify the **IGNORE** option. The reason for this is that the client/server protocol does not allow interrupting the transfer of

the data file from client host to server after the transfer has started, so there is no convenient way to abort the operation in the midd

## **Chapter 9**

### **Programming Inside MySQL**

# Stored Routines

- A stored routine is a named set of SQL statements that is stored on the server:
  - Clients do not need to keep reissuing the individual statements but can refer to the stored routine instead.
- Stored routine types:
  - **Stored procedures:** Invoke procedures with a **CALL** statement. They can pass back values using output variables or result sets.
  - **Stored functions:** Call functions inside a statement. They return scalar values.

## Database Administrators

This lesson focuses on stored procedures from a DBA perspective. A large number of DBAs are tasked with the responsibility of supporting stored procedures rather than actually creating stored procedures.

### ***MySQL Advanced Stored Procedures***

Oracle University offers this two-day course in which you learn how to improve the quality of stored procedures, functions, and triggers in your applications as well as how to debug and optimize them. With a focus on hands-on practice, this instructor-led course is designed to teach you how to maximize the use of stored procedures and how to determine when an application should contain stored procedures and when it should not.

# Uses of Stored Routines

- Centralized client functionality
  - Create a routine in MySQL and make it available to multiple client applications.
- Security
  - Minimal data access
  - Single-location processing
- Performance improvement
  - Clients invoke the stored routine by name, and do not send the entire set of statements contained in the routine.
- Function libraries
  - A set of stored routines becomes a library of functions for the database.

## Client Functionality

With stored routines, you can create a statement or series of statements centrally in the database to be used by multiple client applications written in different programming languages, or that work on different platforms.

## Security

Stored routines provide an outlet for those applications that require the highest level of security. Banks, for example, use stored procedures and functions for all common operations. This provides a consistent and secure environment. Routines can be coded to ensure that each operation is properly logged. In such a setup, applications and users have no access to the database tables directly and can execute only specific stored routines.

## Performance

Stored routines can improve performance because less information needs to be sent between the server and the client. The client invokes the stored routine by name rather than passing through all statements that comprise the stored routine.

## Function Libraries

Stored routines allow for libraries of functions in the database server. These libraries act as an API to the database.

# Stored Routines: Issues

- Increased server load
- Limited development tools
- Limited language functionality and speed
- Limited debugging and profiling capabilities

## **Increased Server Load**

Executing stored routines in the database itself can increase the server load and reduce the performance of the applications. Apply testing and common sense to ensure that the convenience of having logic in the database itself outweighs any performance issues that may arise.

## **Limited Development Tools**

Development tools that support stored routines in MySQL are not as mature or well specified as in more general-purpose programming languages. This limitation can make writing and debugging stored routines a much more difficult process and needs to be considered in the decision process.

## **Limited Language Functionality and Speed**

Even though having logic in the database itself is a very significant advantage in many situations, there are limitations on what can be accomplished in comparison to other programming languages. Stored routines execute in the context of the database and provide good performance when processing lots of data compared to routines in client applications, but client application languages may have stronger, more general-purpose processing, integration, or other library features. You must consider the range of functionality required to ensure that you are using the best possible solution for each routine.

# Executing Stored Routines

- Executing procedures
- Executing functions
- Implications of database association:
  - **USE <database>**
  - Qualify names
  - Routines are deleted when the database is deleted.
- **SELECT** statements:
  - Stored procedures only
  - The result set is sent directly to the client.

The commands for calling stored routines are very similar to other commands in MySQL.

Invoke stored procedures by using a `CALL` statement. Stored procedures pass back values using output variables or result sets.

Invoke functions from inside a statement just like any other function (that is, by invoking the function's name); functions return a scalar value.

Each stored routine is associated with a particular database. This has several implications:

- **USE <database>**: When you invoke a routine, MySQL executes an implicit `USE <database>` for the duration of the routine. You cannot issue a `USE` statement within a stored routine.
- **Qualify names**: You can qualify a routine name with its database name. Do this to refer to a routine that is not in the current database. For example, to invoke a stored procedure `p` or function `f` that is associated with the `test` database, use `CALL test.p()` or `test.f()`.
- **Database deletions**: When you drop a database, all stored routines associated with it are also dropped.

MySQL allows the use of regular `SELECT` statements inside a stored procedure. The result set of such a query is sent directly to the client.

## Stored Procedure: Example

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE record_count ()
-> BEGIN
->     SELECT 'Country count ', COUNT(*)
->         FROM Country;
->     SELECT 'City count ', COUNT(*) FROM
->         City;
->     SELECT 'CountryLanguage count',
->         COUNT(*) FROM CountryLanguage;
-> END//
mysql> DELIMITER ;
```

### Compound Statements

You can use BEGIN...END syntax in stored routines and triggers to create a compound statement. The BEGIN...END block can contain zero or more statements. An empty compound statement is legal, and there is no upper limit on the number of statements within a compound statement.

### Delimiter

Within the BEGIN...END syntax, you must terminate each statement with a semicolon (;). Because the mysql client uses the semicolon as the default terminating character for SQL statements, you must use the DELIMITER statement to change this when using the mysql command-line client interactively or for batch processing.

In the example in the slide, the first DELIMITER statement makes the terminating SQL statement character two forward slashes (//). This change ensures that the client does not interpret the semicolons in the compound statement as statement delimiters, and ensures that the client does not prematurely send the CREATE PROCEDURE statement to the server. When the statement creating the stored routine terminates with //, the client sends the statement to the server before issuing the second DELIMITER statement to reset the statement delimiter to a semicolon.

## Stored Function: Example

```
mysql> DELIMITER //
mysql> CREATE FUNCTION pay_check (gross_pay
->      FLOAT(9,2), tax_rate FLOAT (3,2))
-> RETURNS FLOAT(9,2)
-> NO SQL
-> BEGIN
->     DECLARE net_pay FLOAT(9,2)
->     DEFAULT 0;
->     SET net_pay=gross_pay - gross_pay *
->           tax_rate;
->     RETURN net_pay;
-> END //
mysql> DELIMITER ;
```

### RETURNS Clause

The RETURNS clause identifies the type of value to be returned by this function.

### Characteristics

Several characteristics provide information about the nature of data used by the routine. In MySQL, these characteristics are advisory only. The server does not use them to constrain what kinds of statements a routine is permitted to execute.

- CONTAINS SQL indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics are given explicitly.
- NO SQL indicates that the routine contains no SQL statements.
- READS SQL DATA indicates that the routine contains statements that read data (for example, SELECT) but not statements that write data.
- MODIFIES SQL DATA indicates that the routine contains statements that may write data (for example, INSERT or DELETE).

**Note:** When you create a function with binary logging enabled, MySQL produces an error if you do not specify one of the following: NO SQL, READS SQL DATA, or DETERMINISTIC.

## **DECLARE Statement**

Use the `DECLARE` statement to declare local variables and to initialize user variables in stored routines. You can add the `DEFAULT` clause to the end of the `DECLARE` statement to specify an initial value for the user variable. If you leave out the `DEFAULT` clause, the initial value for the user variable is `NULL`.

## **SET Statement**

The `SET` statement allows you to assign a value to defined variables by using either `=` or `:=` as the assignment operator.

## **RETURN Statement**

The `RETURN` statement terminates execution of a stored function and returns the value expression to the function caller.

# Examine Stored Routines

- **SHOW CREATE PROCEDURE** and **SHOW CREATE FUNCTION**:
  - MySQL-specific
  - Return exact code string
- **SHOW PROCEDURE STATUS** and **SHOW FUNCTION STATUS**:
  - MySQL-specific
  - Return characteristics of routines
- **INFORMATION\_SCHEMA.ROUTINES**:
  - Contains all information displayed by the **SHOW** commands
  - Holds the most complete picture of the stored routines available in all databases

## **SHOW CREATE PROCEDURE** and **SHOW CREATE FUNCTION**

These statements are MySQL extensions and are similar to **SHOW CREATE TABLE**. These statements return the exact string that can be used to re-create the named routine. One of the main limitations of these statements is that you must know the name of the procedure or function and must know whether it is a procedure or function before you can attempt to view the information.

## **SHOW PROCEDURE STATUS** and **SHOW FUNCTION STATUS**

These statements are specific to MySQL. They return characteristics of routines, such as the database, name, type, creator, and creation and modification dates. These statements have the advantage of being able to display specific routines based on a **LIKE** pattern. If no pattern is specified, the information for all stored procedures or all stored functions is listed, depending on which statement is used. For example, the following statement shows information about procedures that have a name beginning with “film”:

```
SHOW PROCEDURE STATUS LIKE 'film%'\G
```

## **INFORMATION\_SCHEMA.ROUTINES**

The INFORMATION\_SCHEMA.ROUTINES table contains information about stored routines (both procedures and functions) and returns the majority of the details that can be found in both the SHOW CREATE ... and SHOW ... STATUS statements to include the actual syntax used to create the stored routines. Of the three options, this table holds the most complete picture of the stored routines available in the databases.

Example:

```
mysql> SELECT routine_name, routine_schema, routine_type, definer
    >   FROM INFORMATION_SCHEMA.ROUTINES
    > WHERE routine_name LIKE 'film%';
+-----+-----+-----+-----+
| routine_name      | routine_schema | routine_type | definer      |
+-----+-----+-----+-----+
| film_in_stock     | sakila        | PROCEDURE    | root@localhost |
| film_not_in_stock | sakila        | PROCEDURE    | root@localhost |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

## **Tables in the mysql System Database Associated with Programming Components**

The mysql system database contains tables that provide information associated with the stored routine features of MySQL. The tables include:

- The mysql.event table, which contains information about the events stored on the MySQL server
- The mysql.proc table, which contains information about the stored procedures and functions on the MySQL server
- The mysql.procs\_priv table, which provides the access control grant details for users in reference to stored procedures

# Stored Routines and Execution Security

| Privilege             | Operations Allowed                  |
|-----------------------|-------------------------------------|
| <b>CREATE ROUTINE</b> | Create stored routines.             |
| <b>ALTER ROUTINE</b>  | Alter or drop stored routines.      |
| <b>EXECUTE</b>        | Execute stored routines.            |
| <b>GRANT OPTION</b>   | Grant privileges to other accounts. |

- Automatically obtain **EXECUTE** and **ALTER ROUTINE** when creating a stored routine.
- **GRANT** examples:

```
mysql> GRANT ALL ON world_innodb.* TO
      -> 'magellan'@'localhost';
mysql> GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE
      -> world_innodb.record_count TO
      -> 'magellan'@'localhost';
```

Several privileges apply to the use of stored procedures and functions.

## Default Actions

When you create a stored routine, MySQL automatically grants the EXECUTE and ALTER ROUTINE privileges to your account for that routine. These privileges can be revoked or removed later by a user who has the privilege being revoked and who also has the GRANT OPTION privilege. You can verify these privileges by issuing a SHOW GRANTS statement after creating a routine.

## Granting Privileges

The GRANT ALL statement, when granting all privileges at a global or database level, includes all stored routine privileges except GRANT OPTION.

To grant the GRANT OPTION privilege, include the WITH GRANT OPTION clause at the end of the statement. You can grant the EXECUTE, ALTER ROUTINE, and GRANT OPTION privileges at the individual routine level, but only for routines that already exist. To grant privileges for an individual routine, qualify the routine with its database name, and provide the keyword PROCEDURE or FUNCTION to indicate the routine type, as in the following example:

```
GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE world_innodb.record_count
    TO 'magellan'@'localhost' WITH GRANT OPTION;
```

## Quiz

You can invoke a stored procedure using a **CALL** statement, and it can only pass back values using output variables or result sets.

- a. True
- b. False

# Triggers

Triggers are named database objects that activate when you modify table data. They can:

- Examine data before it is inserted or updated, and verify deletes and updates
- Act as a data filter by modifying data if it is out of range, before an insert or update
- Modify how **INSERT**, **UPDATE**, and **DELETE** behave
- Mimic the behavior of foreign keys for storage engines that do not support foreign keys
- Provide logging functionality
- Automatically create summary tables

Database triggers are named database objects that are maintained within a database and are activated when data within a table is modified. You can use triggers to bring a level of power and security to the data within the tables.

## Trigger Features

You can use triggers to provide control over access to specific data, the ability to perform specific logging, or auditing of the data itself.

# Create Triggers

- The **CREATE TRIGGER** statement:

```
CREATE TRIGGER trigger_name
    { BEFORE | AFTER }
    { INSERT | UPDATE | DELETE }
    ON table_name
    FOR EACH ROW
    triggered_statement
```

- Example:

```
CREATE TRIGGER City_AD AFTER DELETE
    ON City
    FOR EACH ROW
    INSERT INTO DeletedCity (ID, Name)
    VALUES (OLD.ID, OLD.Name);
```

*trigger\_name* is the name you give the trigger, and *table\_name* is name of the table you want the trigger to be associated with. BEFORE and AFTER indicates when the trigger activates, either before or after the triggering event, and INSERT, UPDATE, or DELETE indicates what that event is.

**Note:** The table names OLD and NEW refer to virtual tables made visible to the trigger. They contain the old version of data modified by UPDATE or DELETE statements, or the new version of data added by INSERT or UPDATE statements, respectively.

# Trigger Events

- Before:
  - **BEFORE INSERT**: Triggered before new data is added
  - **BEFORE UPDATE**: Triggered before existing data is updated (or overwritten) with new data
  - **BEFORE DELETE**: Triggered before data is deleted
- After:
  - **AFTER INSERT**: Triggered after new data is added
  - **AFTER UPDATE**: Triggered after existing data is updated (or overwritten) with new data
  - **AFTER DELETE**: Triggered after data is deleted

## **BEFORE** and **AFTER**

The BEFORE and AFTER keywords refer to the trigger's activation time relative to when the data modification statement (`INSERT`, `UPDATE`, or `DELETE`) writes its changes to the underlying database.

The BEFORE keyword causes the trigger to execute before the data modification in question. Use BEFORE triggers to capture invalid data entries and correct or reject them prior to writing to the table.

The AFTER keyword defines a trigger that executes when the data modification succeeds. Use AFTER triggers to log or audit the modification of data within your databases.

# Trigger Error Handling

MySQL handles errors during trigger execution as follows:

- Failed **BEFORE** triggers
  - The transaction containing the operation on the corresponding row rolls back.
- **AFTER** trigger execution
  - **BEFORE** trigger events and the row operation must execute successfully.
- For non-transactional tables, transactions are not possible.
  - Only the statement that fires the trigger rolls back.

When a trigger fails, MySQL rolls back the transaction containing the statement that causes the trigger to fire.

For non-transactional tables, such rollback cannot be done. As a result, although the statement fails, any changes performed prior to the point of the error remain in effect.

# Examining Triggers

- **SHOW CREATE TRIGGER *trigger\_name*:**
  - MySQL-specific
  - Returns the exact code string
- **SHOW TRIGGERS:**
  - MySQL-specific
  - Returns characteristics of triggers
- **INFORMATION\_SCHEMA.TRIGGERS:**
  - Contains all data displayed by the **SHOW** commands
  - Holds the most complete picture of the triggers available in all databases

## **SHOW CREATE TRIGGER *trigger\_name***

This statement returns the exact string that you can use to re-create the named trigger. You must know the name of the trigger to run this statement; there is no **LIKE** or **WHERE** syntax for the **SHOW CREATE TRIGGER** statement.

## **SHOW TRIGGERS**

This statement is a MySQL extension. It returns characteristics of triggers, such as the database, name, type, creator, and creation and modification dates. This statement has the advantage of being able to display specific triggers based on conditions provided in a **LIKE** pattern or a **WHERE** clause. If you do not specify a condition, the statement displays information for all triggers.

## Dropping Triggers

- Explicitly drop a trigger by using this syntax:

```
DROP TRIGGER [IF EXISTS]
[ schema_name. ]trigger_name;
```

- Use **IF EXISTS** to prevent the error that occurs when you attempt to delete a trigger that does not exist.
- Implicitly drop a trigger by dropping:
  - The table on which the trigger is defined
  - The database containing the trigger

When you use `DROP TRIGGER trigger_name`, the server looks for that trigger name in the current schema. If you want to drop a trigger in another schema, include the schema name.

# Restrictions on Triggers

- Disallowed statements include:
  - SQL-prepared statements
  - Explicit or implicit **COMMIT** and **ROLLBACK**
  - Statements that return a result set
  - **FLUSH** statements
  - Statements that modify the table to which the trigger applies
- Triggers are not fired in response to:
  - Changes caused by cascading foreign keys
  - Changes caused during row-based replication

## Result Sets Cannot Be Returned

The following are not allowed:

- `SELECT` statements that do not have an `INTO var_list` clause
- `SHOW` statements

Process a result set within a trigger either with `SELECT ... INTO var_list` or by using a cursor and `FETCH` statements.

## Trigger Privileges

- To execute the **CREATE TRIGGER** and **DROP TRIGGER** commands, you need the **TRIGGER** privilege.
- You need additional privileges to use **OLD** and **NEW** within a trigger:
  - To assign the value of a column with **SET NEW.col\_name = value**, you need the **UPDATE** privilege for the column.
  - To use **NEW.col\_name** in an expression to refer to the new value of a column, you need the **SELECT** privilege for the column.

## **Quiz**

Triggers are fired in response to the changes that are caused by cascading foreign keys.

- a. True
- b. False

# Events

- An *event* is a named database object that contains SQL statements and executes according to a schedule.
  - Begins and ends at a specific date and time
  - Similar to a UNIX **crontab** or Windows task scheduler

```
CREATE EVENT event_name
ON SCHEDULE schedule DO sql_statement
```

- Event statements:
  - **SET GLOBAL event\_scheduler = {ON | OFF}**
  - **CREATE EVENT**
  - **ALTER EVENT**
  - **DROP EVENT**

MySQL events are tasks that run according to a schedule. They can be referred to as “scheduled events.” When you create an event, it is created as a named database object containing a SQL statement (or stored procedure) that executes at a certain moment in time, optionally recurring at a regular interval. Conceptually, this is similar to the idea of the Linux/UNIX **crontab** (also known as a “cron job”) or the Windows Task Scheduler.

## Mandatory items

**event\_name:** Events are schema objects, just like tables, stored procedures, and triggers. The *event\_name* must be a valid identifier, and it may be quoted and/or qualified with the schema name in the usual way. Event names must be unique within the schema.

**schedule:** The schedule is a rule that specifies when MySQL executes the action that is associated with the event.

**sql\_statement:** You must include a valid SQL statement or stored procedure statement that is executed according to the schedule. This statement is subject to the same limitations applied to stored functions and dynamic SQL. For example, the statement cannot return a result set (such as those produced by **SELECT** or **SHOW**). Typically, you use a **CALL** statement to call a procedure to perform the actual action.

## **Event Scheduler**

When you create an event, it is stored in the database so it can be executed according to a schedule. The schedules of all events are monitored by the `event_scheduler` thread, which starts a new thread to execute each event when the scheduled time arrives.

By default, the `event_scheduler` thread is set to `OFF`. You must explicitly enable it by modifying the value of the global `event_scheduler` server variable, setting it to `ON`. You can do this either by adding the server variable to an option file (so the change takes effect on startup) or by dynamically using the `SET` syntax. If you start the server with `event_scheduler` set to `DISABLED`, you cannot enable it with a `SET` statement while MySQL is running. Rather, you must stop MySQL and restart it with the option enabled.

After enabling the `event_scheduler` thread, you can see it in the output of `SHOW PROCESSLIST` (and the `INFORMATION_SCHEMA` equivalent, `PROCESSLIST`).

## **Altering Events**

You can change `EVENTS` by using the `ALTER EVENT` syntax. You can change each element of the `EVENT` with this syntax. Consequently, the syntax model for `ALTER EVENT` is almost identical to that of the `CREATE EVENT` statement.

# Schedule

- A *schedule* is a rule that specifies when an event executes.
- There are two types of schedules:
  - Executed once (using the **AT** keyword)
  - Executed repeatedly (using the **EVERY** keyword)
- The Event Scheduler executes scheduled events
  - Launches a separate thread to execute each event
- Use events for automatic, periodic execution of tasks.
- Example:
  - Regular CSV dumps of table data
  - ANALYZE TABLE
  - Creating periodic summary tables

A schedule is a rule that specifies when an action should be executed. You can specify the schedule in the **SCHEDULE** clause of the **CREATE EVENT** and **ALTER EVENT** statements.

## Types of Schedule Actions

There are two types of schedules: those that are executed once (using the **AT** keyword) and those that can be executed repeatedly (using the **EVERY** keyword). For the latter, you must define the frequency of the event's repetition. You can also define a time window that determines the period during which the event should be repeated.

The syntax for the **SCHEDULE** clause is as follows:

```
AT timestamp [+ INTERVAL interval] | EVERY interval [STARTS  
timestamp [+ INTERVAL interval]] [ENDS timestamp [+ INTERVAL  
interval]]
```

## SCHEDULE Clause

The SCHEDULE clause can contain the following variable elements:

- *timestamp*: An expression of the DATETIME or TIMESTAMP type
- *interval*: Specifies a duration. The duration is expressed by specifying a quantity as an integer followed by a keyword that defines a particular kind of duration. Valid keywords are:
  - YEAR
  - QUARTER
  - MONTH
  - DAY
  - HOUR
  - MINUTE
  - WEEK
  - SECOND
  - YEAR\_MONTH
  - DAY\_HOUR
  - DAY\_MINUTE
  - DAY\_SECOND
  - HOUR\_MINUTE
  - HOUR\_SECOND
  - MINUTE\_SECOND

## Event Scheduler

The Event Scheduler, which is a separate thread in the mysqld process, is responsible for executing scheduled events. It checks whether it should execute an event; if it should, it creates a new connection to execute the action.

## Event Scheduler Use Cases

Use events for automatic, periodic execution of (maintenance) tasks, such as updating a summary table or refreshing a table from a query (materialized view emulation), or for performing nightly jobs. Examples include processing the day's work, loading a data warehouse, or exporting data to file.

# Event Scheduler and Privileges

- You must have the **SUPER** privilege to set the global **event\_scheduler** variable.
- You must have the **EVENT** privilege to create, modify, or delete events.
- Use **GRANT** to assign privilege (at the schema level only):

```
mysql> GRANT EVENT ON myschema.* TO user1@srv1;
mysql> GRANT EVENT ON *.* TO user1@srv1;
```

- Use **REVOKE** to cancel an event privilege:

```
REVOKE EVENT ON myschema.* FROM user1@srv1;
```

## Revoke Event Privilege

To cancel the **EVENT** privilege, use the **REVOKE** statement. Revoking the **EVENT** privilege from a user account does not delete or disable any events already created by that account.

### mysql Tables

Users' **EVENT** privileges are stored in the **Event\_priv** columns of the **mysql.user** and **mysql.db** tables. In both cases, this column holds one of the values '**Y**' or '**N**'. '**N**' is the default value.

The value of **mysql.user.Event\_priv** is set to '**Y**' for a given user only if that user has the global **EVENT** privilege.

For a schema-level **EVENT** privilege, **GRANT** creates a row in **mysql.db** and sets that row's column values as follows:

- **Db**: The name of the schema
- **User**: The name of the user
- **Event\_priv**: '**Y**'

You do not have to manipulate these tables directly, because the **GRANT EVENT** and **REVOKE EVENT** statements perform the required operations on them.

# Event Execution Privileges

- An event executes with the privileges of the event definer.
  - It cannot perform a task for which the definer does not have the privilege.
- Example
  - `user1@srv1` can create a `SELECT` event for `myschema` only:

```
CREATE EVENT e_store_ts
  ON SCHEDULE
    EVERY 10 SECOND
  DO
    INSERT INTO myschema.mytable VALUES
      (UNIX_TIMESTAMP());
```

- The event does not create any rows in the table due to lack of `INSERT` privilege.

It is important to understand that an event is executed with the privileges of its definer, and that it cannot perform any actions for which its definer does not have the required privileges. For example, suppose that `user1@srv1` has the `EVENT` privilege for `myschema`. Suppose also that he has the `SELECT` privilege for `myschema`, but no other privileges for this schema. Even though it is possible for `user1@srv1` to create a new event, the event itself cannot perform an `INSERT` operation because the definer, `user1@srv1`, does not have the privileges to perform the action.

## Error Log

If you inspect the MySQL error log (`hostname.err`), you can see that the event is executing but that the action it is attempting to perform fails, as indicated by `RetCode=0`:

```
...
060209 22:39:44 [Note] EVEX EXECUTING event newdb.e [EXPR:10]
060209 22:39:44 [Note] EVEX EXECUTED event newdb.e [EXPR:10]. RetCode=0
060209 22:39:54 [Note] EVEX EXECUTING event newdb.e [EXPR:10]
060209 22:39:54 [Note] EVEX EXECUTED event newdb.e [EXPR:10]. RetCode=0
060209 22:40:04 [Note] EVEX EXECUTING event newdb.e [EXPR:10]
060209 22:40:04 [Note] EVEX EXECUTED event newdb.e [EXPR:10]. RetCode=0
...
...
```

# Examining Events

- **SHOW CREATE EVENT *event\_name***
  - MySQL-specific
  - Returns a string containing the code needed to re-create the event
- **SHOW EVENTS**
  - MySQL-specific
  - Returns the characteristics of events
- **INFORMATION\_SCHEMA.EVENTS**

|   |  |  |  |  |  |  |  |  |
|---|--|--|--|--|--|--|--|--|
| EVENT_CATALOG: NULL                                   |  |  |  |  |  |  |  |  |
| EVENT_SCHEMA: myschema                                |  |  |  |  |  |  |  |  |
| EVENT_NAME: e_store_ts                                |  |  |  |  |  |  |  |  |
| DEFINER: user1@srvl                                   |  |  |  |  |  |  |  |  |
| EVENT_BODY: SQL                                       |  |  |  |  |  |  |  |  |
| EVENT_DEFINITION: INSERT INTO myschema.mytable VALUES |  |  |  |  |  |  |  |  |
| (UNIX_TIMESTAMP())                                    |  |  |  |  |  |  |  |  |
| ...   |  |  |  |  |  |  |  |  |

## SHOW CREATE EVENT *event\_name*

This statement displays the CREATE EVENT statement needed to re-create a given event. You must provide the name of the event to view information about that event.

## SHOW EVENTS

This statement is a MySQL extension. It returns characteristics of events, such as the database, name, type, creator, and creation and modification dates. This statement has the advantage of being able to display specific events based on a condition provided in a LIKE pattern or WHERE clause. If you do not provide such a condition, the SHOW EVENTS statement shows the information for all events.

## Dropping Events

Explicitly drop an event by using this syntax:

```
DROP EVENT [ IF EXISTS ]  
[ schema_name.]event_name;
```

- Use **IF EXISTS** to prevent the error that occurs when you attempt to delete an event that does not exist.
- You must have the **EVENT** privilege for the database that contains the event to be dropped.

## **Quiz**

The schedules of all events are monitored by which thread?

- a. activate\_events thread
- b. event\_scheduler thread
- c. monitor\_events thread
- d. schedule\_events thread

## SIGNAL and RESIGNAL

- MySQL supports the use of **SIGNAL** and **RESIGNAL**:
  - Can be used to raise a specific SQLSTATE
  - Similar to exception handling in other languages
  - Used for advanced flow control when handling errors
- Use **SIGNAL** to throw an error or warning state.
- Use **RESIGNAL** to forward an error or warning state originated by a prior SIGNAL.

These commands provide a way for stored routines and triggers to return an error to application programs or the end user. SIGNAL provides error information to a handler, to an outer portion of the application, or to the client. For example, use SIGNAL within a stored procedure to throw an error to the code that called the stored procedure. That code can handle the error in an appropriate way.

RESIGNAL passes on the error condition information that is available during execution of a condition handler. Use RESIGNAL within a compound statement inside a stored procedure or function, trigger, or event. For example, if stored procedure *p* calls stored procedure *q*, and if *q* uses SIGNAL to throw an error, *p* can declare a handler to process signals from *q*. The handler in *p* can use the RESIGNAL statement to throw the same error information to the code that calls *p*.

## **Chapter 10**

### **Backup and Recovery**

# **Objectives**

After completing this lesson, you should be able to:

- Describe backup basics
- List the types of backups
- List MySQL backup tools and utilities
- Make binary and text backups
- Explain the role of log and status files in backups
- Perform data recovery

# Backup Basics

- Most important reasons to have backups:
  - Complete system recovery
  - Auditing capabilities
  - Typical DBA tasks
- Backup types:
  - Hot
  - Cold
  - Warm

## Reasons for Backups

- **Complete system recovery:** If a system fails, it is crucial to have a backup of the system so you can restore it. The backup and recovery strategy that you implement is determined by how complete and how current the recovered data needs to be.
- **Auditing capabilities:** For certain systems and associated processes, you might need to audit or analyze the data in an environment separate from the primary production environment. You can use backups to create such a separate environment.
- **Typical DBA tasks:** Use backups when you need to perform typical DBA tasks such as transferring data from one system to another, creating a development server based on a specific state of a production server, or recovering certain portions of the system to a state prior to a user error.

- **Hot:** These dynamic backups take place while the data is being read and modified with little to no interruption of your ability to interface with or manipulate data. With hot backups, the system remains accessible for operations that read and modify data.
- **Cold:** These backups take place while the data is inaccessible to users, so that you cannot read or make any modifications to the data. These offline backups prevent you from performing any activities with the data. These types of backups do not interfere with the performance of the system when it is up and running. However, having to lock out or completely block user access to data for an extended period of time is not acceptable for some applications.
- **Warm:** These backups take place while the data is being read, but in most cases the data itself cannot be modified while the backup is taking place. This middle-of-the-road backup type has the advantage of not having to completely lock out end users. However, the disadvantage of not being able to modify the data sets while the backup is taking place can make this type of backup not reasonable for certain applications. Not being able to modify data during backups can produce performance issues.

# Backup Basics



Disk



Binary logs



Logical/textual backups

- **Disk:** You can back up data directly to another disk by using processes such as replication or RAID mirroring, or by using external applications such as DRBD. These techniques provide live (or nearly live) backups and a quick means of recovering data.
- **Binary logs:** A binary log records modifications to the data. The binary logs are therefore very useful for restoring those events that have occurred since the last full backup. The advantage to backing up binary logs is that they contain a record of all changes to the data over time, rather than a snapshot of the data. You can take multiple binary log backups in sequential order. Decide on the number of binary log backups to take between backups based on how much data is being modified and how often a full backup is completed. The disadvantage to binary logs is you must restore all sequential binary logs that were created from the last full backup in sequence. In addition, recovery from a system failure can be slow depending on the number of binary logs that must be restored.

- **Logical/textual backup:** You can do a complete data dump by using `mysqldump`. These data dumps are based on a specific point in time but are the slowest of all the backup copies. The advantage to using `mysqldump` is that the file that is created is simply a SQL script containing statements that you can run on a MySQL server. The disadvantage is that `mysqldump` locks tables during the dump, which prevents users from reading or writing to the files during the backup.

## **Backups with MySQL**

- Logical (textual representation: SQL statements)
- Physical (binary copy of data files)
- Snapshot-based
- Replication-based
- Incremental (binary log flushed)

MySQL backups can be one of the following:

- A logical backup that results in a text file containing SQL statements to reconstruct the database
- A physical backup, which is a binary copy of the MySQL database files
- A snapshot-based backup
- A replication-based backup
- An incremental backup (created by flushing the MySQL binary logs)

## **Quiz**

Which backup type takes place while the data is being read and modified with little to no interruption to your ability to interface or manipulate data?

- a. Cold backup
- b. Hot backup
- c. Warm backup

# Logical (Textual) Backups

- Logical backups:
  - Convert databases and tables to SQL statements
  - Are portable
  - Require that the MySQL server be running during the backup
  - Can back up both local and remote MySQL servers
  - Are generally slower than raw (binary) backups
- The size of a logical backup file may exceed the size of the database being backed up.

## SQL Statements

Logical (or textual) backups dump the contents of the database into text files. These text files contain SQL statements and, as such, are very portable. These SQL statements contain all the information needed to rebuild the MySQL databases and tables. You can use the text file to reload the database on another host, running a different architecture.

## Server Must Be Running

The MySQL server must be running when you create logical backups because the server creates the files by reading the structure and contents of the tables being backed up, and then converting the structure and data to SQL statements. Other applications can perform read operations during the logical backup.

## Local and Remote Servers

With logical backups, you can back up local and remote MySQL servers. You can perform other types of backup (raw) only on the local MySQL server.

## **Generally Slower**

Logical backups are generally slower than raw backups. This is because the MySQL server must read tables and interpret their contents. It must then translate the table contents to a disk file, or send the statements to a client program, which writes them out.

A logical backup is also slower than a raw backup during recovery. This is because the method of recovery is to execute the individual `CREATE` and `INSERT` statements to re-create each backed-up table and row.

Logical backups may exceed the size of the database being backed up, because the statements themselves may take up more space than the data they represent. However, because InnoDB stores data in data pages, which can contain free space, the InnoDB data files often take considerably more space on disk than the data requires.

# Physical (Raw or Binary) Backups

- Physical backups:
  - Make an exact copy of the database files
    - You can use standard commands such as `tar`, `cp`, `cpio`, `rsync`, or `xcopy`.
  - Must be restored to the same database engine
  - Can be restored across machine architectures
  - Are faster than logical backups and recoveries
- Database files must not change during backup.
  - The method to achieve this depends on the storage engine.
  - For InnoDB: MySQL server shutdown is required.
  - For MyISAM: Lock tables to allow reads but not changes.
  - Snapshot, replication, or proprietary methods can be used.

## Exact Copy

Raw backups are binary copies of the MySQL database files. These copies preserve the databases in exactly the same format in which MySQL itself stores them on disk. Because they are exact copies of the originals, raw backups are the same size as the original.

Raw binary backups are faster than logical backups because the process is a simple file copy, which does not need to know anything about the internal structure of the files. However, if you use a raw backup to transfer databases to another machine with a different architecture, the files must be binary portable. Because a raw backup is an exact representation of the bits in the database files, you must restore them to a MySQL server using the same database engine. When you recover a raw MySQL backup from an InnoDB table, it remains an InnoDB table on the target server.

## MySQL Server Shutdown

With binary backup methods, you must make sure that the server does not modify the files while the backup is in progress. This can be accomplished in a variety of ways. One way is to shut down the MySQL server and then take the backup. This has obvious disadvantages. For some storage engines, a better approach is to temporarily lock the database, take the backup, and then unlock the database. You can also minimize the effect to MySQL and applications by using snapshots, replication, or proprietary methods.

# Snapshot-Based Backups

Snapshot-based backups:

- Create a point-in-time “copy” of the data
- Provide a logically frozen version of the file system from which you can take MySQL backups
- Greatly reduce the time during which the database and applications are unavailable

A raw backup is typically performed against the snapshot copy.

## External Snapshot Capability

Snapshot-based backups use a snapshot capability that is external to MySQL. For example, if your MySQL databases and binary logs exist on an LVM2 logical volume with an appropriate file system, you can create snapshot backups. Snapshot-based backups work best for transactional engines such as InnoDB. You can perform a hot backup of InnoDB tables; for other engines, you can perform a warm backup.

## Scalable

Snapshot backups are scalable, because the time needed to perform the snapshot does not increase as the size of the database grows. In fact, the backup window (from the perspective of the application) is almost zero. However, a snapshot-based backup almost always includes a raw backup after the snapshot is taken.

# Replication-Based Backups

- MySQL replication can be used for backups:
  - The master is used for the production applications.
  - A slave is used for backup purposes.
- This eliminates the impact on production applications.
- The backup of the slave is either logical or raw.
- Higher cost: There must be another server and storage to store replica of the database.
- Based on asynchronous replication:
  - The slave may be delayed with respect to the master.
  - This is acceptable if the binary log is not purged before the slave has read it.

## One-Way, Asynchronous Replication

MySQL features support for one-way asynchronous replication, in which one server acts as the master while one or more other servers act as slaves. You can use replication to perform backups by using the replica or slave instead of the master. This has the advantage that the backup operation does not impact the performance of the master server.

## Disadvantages

This is a more expensive solution, since you must purchase additional hardware and network bandwidth. In addition, the slave's copy of the database contains a delayed version of the data relative to the master because of the inherent latency of replication.

## **Quiz**

Which backup type converts databases and tables to SQL statements?

- a. Logical backup
- b. Physical backup
- c. Replication-based backup
- d. Snapshot-based backup

# Binary Logging and Incremental Backups

- Control binary logging at the session level:

```
SET SQL_LOG_BIN = {0|1|ON|OFF}
```

- Flush binary logs when taking logical or raw backups.
  - Synchronize binary log to backups.
- Logical and raw backups are full backups.
  - All rows of all tables are backed up.
- To perform an incremental backup, copy binary logs.
- Binary logs can be used for fine-granularity restores.
  - You can identify transactions that caused damage and skip them during restore.

## Changes Made After Backup

A backup is only one of the components you need for data recovery after loss or damage. The other is the binary log, which contains a record of data changes. To recover databases, use the backups to restore them to their state at backup time. After the backup has been restored, apply the contents of the binary log to apply all data changes since the backup was created. Make sure that binary logging is enabled for all of the MySQL servers.

### SUPER Privilege

You must have the SUPER privilege to set this variable. If you attempt to set this variable without the SUPER privilege, you get the following error:

```
ERROR 1227 (42000): Access denied; you need the SUPER privilege for this operation
```

# Backup Tools: Overview

- SQL statements for logical backups
- SQL statements combined with operating system commands for raw backup
- Other raw backup tools for MySQL:
  - MySQL Enterprise Backup
  - **mysqldump**: Logical backups
- Third-party tools

Backups can be made without the use of additional tools or utilities. Examples include:

- SQL statements that are used to perform logical backups
- Performing raw backups of a combination of SQL statements (for locking) along with operating system commands (for making the binary copy)

## MySQL Enterprise Backup

The MySQL Enterprise Backup product performs hot backup operations for MySQL databases. The product is architected for efficient and reliable backups of tables created by the InnoDB storage engine. For completeness, it can also back up tables from other storage engines.

You can find more information about MySQL Enterprise Backup at:

<http://dev.mysql.com/doc/mysql-enterprise-backup/3.8/en/intro.html>

### **mysqldump**

The `mysqldump` utility comes with the MySQL distribution. It performs logical backups and works with any database engine. It can be automated with the use of `crontab` in Linux and UNIX, and with the Windows Task Scheduler in Windows. There are no tracking or reporting tools for `mysqldump`.

### **mysqlhotcopy**

The `mysqlhotcopy` utility also comes with the MySQL distribution. It performs a raw backup and is used for only those databases that use the MyISAM or ARCHIVE database engines. The name implies that `mysqlhotcopy` performs a “hot” backup, meaning that there is no interruption to database availability. However, because the database is locked for reading and cannot be changed during the backup, it is best described as a “warm” backup. There is no reporting or tracking provided with this script.

## **Third-Party Tools**

Oracle commercial and community tools are the primary focus of this lesson. There are commercial and community third-party tools that you can incorporate into your backup strategies.

# MySQL Enterprise Backup

- Hot backup
  - InnoDB storage engine
- Warm backup
  - Non-InnoDB storage engines
- Incremental backups:
  - Back up data that changed since the previous backup
  - Are primarily intended for InnoDB tables, or non-InnoDB tables that are read-only or rarely updated
- Single-file backups:
  - Provide the ability to create a backup in a single-file format
  - Can be streamed or piped to another process

## Hot Backup

Hot backups are performed while the database is running. This type of backup does not block normal database operations, and it captures even changes that occur while the backup is happening.

`mysqlbackup` is the command-line tool of the MySQL Enterprise Backup product. For InnoDB tables, this tool performs a hot backup operation.

## Warm Backup

For non-InnoDB storage engines, MySQL Enterprise Backup performs a warm backup in which the database tables can be read, but the database cannot be modified while the non-InnoDB backup runs.

## Single-File Backup

Because the single-file backup can be streamed or piped to another process, such as a tape backup or a command such as `scp`, you can use this technique to put the backup on another storage device or server without significant storage overhead on the original database server.

# MySQL Enterprise Backup

Raw files that are backed up with `mysqlbackup`

- InnoDB data
  - `ibdata*` files: Shared tablespace files
  - `.ibd` files: Per-table data files
  - `ib_logfile*` files: Log files
- All files in the data directory to include:
  - `.opt` files: Database configuration information
  - `.TRG` files: Trigger parameters
  - `.MYD` files: MyISAM data files
  - `.MYI` files: MyISAM index files
  - `.FRM` files: Table data dictionary files

## `mysqlbackup` Raw Files

The InnoDB-related data files that are backed up include:

- `ibdata*` files that represent the system tablespace and possibly the data for some user tables
- `.ibd` files, which contain data from user tables
- Data extracted from the `ib_logfile*` files (the redo log information representing changes that occur while the backup is running), which is stored in a new backup file named `ibbackup_logfile`

By default, `mysqlbackup` backs up all files in the data directory. If you specify the `--only-known-file-types` option, the backup includes only additional files with MySQL recognized extensions.

## **mysqlbackup**

- **mysqlbackup** is an easy-to-use tool for all backup and restore operations.
- Basic usage:

```
mysqlbackup -u<user> -p<password>
--backup_dir=<backup-dir>
backup-and-apply-log
```

- **backup**: Performs the initial phase of a backup
- **backup-and-apply-log**: Includes the initial phase of the backup and the second phase, which brings the InnoDB tables in the backup up to date, including any changes made to the data while the backup was running

You can use `mysqlbackup` to take an online backup of your InnoDB tables and a snapshot of your MyISAM tables that correspond to the same binlog position as the InnoDB backup. In addition to creating backups, `mysqlbackup` can pack and unpack backup data, apply to the backup data any changes to InnoDB tables that occurred during the backup operation, and copy data, index, and log files back to their original locations.

### **Backup Procedure**

1. `mysqlbackup` opens a connection to the MySQL server to perform the backups.
2. `mysqlbackup` then takes an online backup of InnoDB tables.
  - This phase does not disturb normal database processing.
3. When the `mysqlbackup` run has almost completed, it executes the SQL command `FLUSH TABLES WITH READ LOCK`, and then it copies the non-InnoDB files (such as MyISAM tables and `.frm` files) to the backup directory.
  - If you do not run long `SELECT` or other queries in the database at this time, and your MyISAM tables are small, the locked phase lasts only a couple of seconds. Otherwise, the whole database, including InnoDB type tables, can be locked until all long-running queries that started before the backup have completed.
4. `mysqlbackup` runs to completion and `UNLOCKS` the tables.

# Restoring a Backup with mysqlbackup

Basic usage:

```
mysqlbackup --backup-dir=<backup-dir>
copy-back
```

- **<backup-dir>**: Specifies where the backed up files are stored
- **copy-back**: Tells **mysqlbackup** to perform a restore operation

The restore operation copies the contents of **<backup-dir>**, including InnoDB and MyISAM indexes, and **.frm** files to their original locations (defined by the **<cnf-file>** file).

## Using the copy-back Option

You must shut down the database server before using **mysqlbackup** with the **copy-back** option. Using this option copies the data files, logs, and other backed-up files from the backup directory back to their original locations, and performs any required post-processing on them.

During the **copy-back** process, **mysqlbackup** cannot query its settings from the server, so it reads the standard configuration files for options such as the **datadir**. If you want to restore to a different server, you can provide a non-standard defaults file with the **--defaults-file** option.

## mysqlbackup Single-File Backups

- Basic usage:

```
mysqlbackup -u<user> -p<password>
--backup-image=<image-file>
--backup_dir=<backup-dir> backup-to-image
```

- Other scenarios

- Standard output:

```
... --backup-dir=<backup-dir> --backup-image=-
backup-to-image > <image-file>
```

- Convert an existing backup directory to a single file:

```
... --backup-dir=<backup-dir>
--backup-image=<image-file>
backup-dir-to-image
```

## Restoring mysqlbackup Single-File Backups

- Extract a selection of files:

```
mysqlbackup -u<user> -p<password>
--backup-image=<image-file>
--backup_dir=<backup-dir> image-to-backup-dir
```

- Other scenarios

- List contents:

```
... --backup-image=<image-file> list-image
```

- Convert an existing backup directory to a single file:

```
... --backup-image=<image_file>
--src-entry=<file-to-extract>
--dst-entry=<file-to-extract> extract
```

- **--src-entry:** Identifies a file or directory to extract from a single-file backup
- **--dst-entry:** Is used with single-file backups to extract a single file or directory to a user-specified path

## Privileges Required for mysqlbackup

- Backup operations require certain privileges
  - Use an existing root account
  - Or, create a new account with sufficient privileges
- Minimum privileges required:
  - **RELOAD** on all databases and tables
  - **CREATE TEMPORARY TABLES** on **mysql** database
  - **CREATE, INSERT, and DROP** on tables:
    - **mysql.ibbackup\_binlog\_marker**
    - **mysql.backup\_progress**
    - **mysql.backup\_history**
  - **SELECT** on **mysql.backup\_history** table
  - **REPLICATION CLIENT** to retrieve the binlog position stored with the backup
  - **SUPER** to optimize locking and minimize disruption

The mysqlbackup client connects to the server with the `--user` and `--password` options. Specifying login details at the command line is not ideal, so consider including them in the [client] section of the mysqlbackup my.cnf file (or other configuration file). The account the connection uses needs certain privileges, as shown in the slide. The following script sets these permissions for the user backupuser:

```
GRANT RELOAD ON *.* TO 'backupuser'@'localhost';
GRANT CREATE TEMPORARY TABLES ON mysql.* TO
'backupuser'@'localhost';
GRANT CREATE, INSERT, DROP, UPDATE ON mysql.ibbackup_binlog_marker
TO 'backupuser'@'localhost';
GRANT CREATE, INSERT, DROP, UPDATE ON mysql.backup_progress TO
'backupuser'@'localhost';
GRANT CREATE, INSERT, SELECT, DROP, UPDATE ON mysql.backup_history
TO 'backupuser'@'localhost';
GRANT REPLICATION CLIENT ON *.* TO 'backupuser'@'localhost';
GRANT SUPER ON *.* TO 'backupuser'@'localhost';
FLUSH PRIVILEGES;
```

## Quiz

Which raw files are backed up with the `mysqlbackup` command?

- a. `ibdata*` files
- b. `.ibd` files
- c. `ib_logfile*` files
- d. All of the above

## **mysqlhotcopy**

- Perl script:
  - Backs up MyISAM and ARCHIVE tables
  - Uses **FLUSH TABLES**, **LOCK TABLES**, and **cp** or **scp** to make a database backup
  - Runs on the same machine where the database directories are located
  - UNIX only
- Basic usage:

```
mysqlhotcopy -u<user> -p<password> <db_name>
               <backup-dir>
```

- Options:
  - **--flush-log**
  - **--record\_log\_pos**

### **LOCK TABLES**

`mysqlhotcopy` connects to the local MySQL server and copies the table files. When it has finished the copy operation, it unlocks the tables.

### **MySQL Server Must Be Running**

Run `mysqlhotcopy` on the server host so that it can copy table files while the table locks are in place. The server must be running so that `mysqlhotcopy` can connect to the server. Operation of `mysqlhotcopy` is fast because it copies table files directly rather than backing them up over the network.

### **Options**

**--flush-log**: Flushes logs after all tables are locked

**--record\_log\_pos = db\_name.tbl\_name**: Records master and slave status in the specified database *db\_name* and table *tbl\_name*

# Raw InnoDB Backups

Backup procedure:

- Execute **FLUSH TABLES...FOR EXPORT**
  - Provide the names of all tables in the command.
- Make a copy of each component:
  - A **.frm** file for each InnoDB table
  - A **.cfg** file for each InnoDB table
  - Tablespace files
    - System tablespace
    - Per-table tablespaces
  - InnoDB log files
  - **my.cnf** file
- Execute **UNLOCK TABLES** to release all table locks.

## Complete InnoDB backup

A raw backup operation that makes a complete InnoDB backup (a backup of all InnoDB tables) is based on making exact copies of all files that InnoDB uses to manage tablespaces. All InnoDB tables in all databases must be backed up together, because InnoDB maintains some information centrally, in the system tablespace.

The `FLUSH TABLES...FOR EXPORT` command ensures that all named tables are in a state ready for copying at the file system level. If you are backing up multiple tables at the same time, you must specify the names of all tables at the same time, for example:

```
FLUSH TABLES City, Country, CountryLanguage FOR EXPORT;
```

The command creates a `.cfg` file for each flushed table. This file allows you to import the table on another server that does not already contain the metadata for that table in the shared tablespace.

InnoDB locks the tables so that other processes cannot write to them during the export process. To release those locks, issue the `UNLOCK TABLES` command after you have copied the required files.

## Recovery

To recover InnoDB tables by using a raw backup, stop the server, replace all the components of which copies were made during the backup procedure, and restart the server.

**Note:** InnoDB stores table metadata in the shared tablespace, so you must either copy the shared tablespace and per-table tablespace files as a group (replacing the target shared tablespace) or import exported per-table tablespaces individually by using `ALTER TABLE . . . IMPORT TABLESPACE`.

# Raw MyISAM and ARCHIVE Backups

Backup procedure:

- While the server is running, lock the table to be copied:

```
mysql> USE mysql  
mysql> FLUSH TABLES users WITH READ LOCK;
```

- Perform a file system copy.
- Start a new binary log file:

```
FLUSH LOGS;
```

- Release the lock after the file system copy:

```
UNLOCK TABLES;
```

## Locked Tables

To make a raw backup of a MyISAM or ARCHIVE table, copy the files that MySQL uses to represent the table. For MyISAM, these are the .frm, .MYD, and .MYI files. For ARCHIVE tables, these are the .frm and .ARZ files. During this copy operation, other programs (including the server) must not be using the table. To avoid server interaction problems, stop the server during the copy operation.

**Note:** Locking tables instead of shutting down the server works on Linux systems. On Windows, file-locking behavior is such that it might not be able to copy table files for tables that are locked by the server. In that case, stop the server before copying table files.

## New Binary Log File

To start a new binary log file, use FLUSH LOGS (before UNLOCK TABLES). As an alternative, SHOW MASTER STATUS returns the current binary log file name and position. The new binary log file contains all statements that change data after the backup (and any subsequent binary log files).

## Recovery

To recover a MyISAM or ARCHIVE table from a raw backup, stop the server, copy the backup table files into the appropriate database directory, and restart the server.

# LVM Snapshots

Perform raw backups using LVM snapshots when:

- The host supports LVM
  - Example: Linux supports LVM2.
- The file system containing the MySQL data directory is on a **logical volume**

Backup procedure:

- Take a **snapshot** of the logical volume containing MySQL's data directory.
  - Use **FLUSH TABLES WITH READ LOCK** if you are backing up non-InnoDB tables.
- Perform a raw backup from the snapshot.
- Remove the snapshot.

On systems that support LVM (such as Linux), you can create a logical volume to contain MySQL's data directory. You can create a snapshot of that volume, and the snapshot behaves like an instantaneous copy of the logical volume. LVM uses a mechanism known as "copy-on-write" to create snapshots that initially contain no data. When you read files from a newly created snapshot, LVM reads those files from the original volume. As the original volume changes, LVM copies data to the snapshot immediately before it changes on the original, so that any data that has changed since taking the snapshot is stored in its original form in the snapshot. The result is that when you read files from the snapshot, it delivers the version of data existing at the instant the snapshot was created.

Because a snapshot is almost instantaneous, you can assume that no changes to the underlying data take place during the snapshot. This makes snapshots very useful for backing up InnoDB databases without shutting the server down.

To create a snapshot, use the following syntax:

```
lvcreate -s -n <snapshot-name> -L <size> <original-volume>
```

The option **-s** instructs `lvcreate` to create a snapshot, with the other options specifying the new snapshot's name and size, and the original volume's location.

For example, assuming a volume group VG\_MYSQL and a logical volume lv\_datadir:

```
lvcreate -s -n lv_datadirbackup -L 2G /dev/VG_MYSQL/lv_datadir
```

The preceding statement creates a snapshot called lv\_datadirbackup with a reserved size of 2 GB. If you only need the snapshot for a short while, the reserved size can be much less than the size of the original volume, because the snapshot's storage contains only those blocks that change in the original. For example, if you are using the snapshot to perform a backup, then it stores only those changes made during the time it takes to perform the backup and drop the snapshot.

You can mount the snapshot as if it were a standard volume. Having mounted it, perform a raw backup from that volume as with any other raw backup (for example, by using `tar` or `cp`). As you are backing up from a snapshot, the database cannot change during the backup process. You are sure to get a consistent version of the data files as they existed at the time of the backup, without needing to stop the server.

Over time, the snapshot's space requirement tends to grow to the size of the original volume. Also, each initial data change to blocks on the original volume causes two data writes to the volume group: the requested change and the copy-on-write to the snapshot. This can affect performance while the snapshot remains. For these reasons, you should remove the snapshot as soon as you have performed the backup. To remove the snapshot created by the preceding statement, use the following statement:

```
lvremove VG_MYSQL/lv_datadirbackup
```

# Raw Binary Portability

- Binary databases can be copied from one MySQL server to another.
- InnoDB
  - All tablespace and log files for the database can be copied directly. The database directory name must be the same on the source and destination systems.
- MyISAM, ARCHIVE
  - All files for an individual table can be directly copied.
- Windows compatibility
  - The MySQL server internally stores lowercase database and table names on Windows systems.
  - For case-sensitive file systems, use an option file statement:  
`lower_case_table_names=1`

## Portability

Binary portability is useful when taking a binary backup made on one machine to use on a second machine that has a different architecture. For example, using a binary backup is one way to copy databases from one MySQL server to another.

### InnoDB

For InnoDB, with binary portability you can copy tablespace files directly from a MySQL server on one machine to another server on another machine, and the second server accesses the tablespace.

For more information about InnoDB portability, see:

<http://dev.mysql.com/doc/mysql/en/innodb-migration.html>

### MyISAM and ARCHIVE

For MyISAM and ARCHIVE, binary portability means that the files can be directly copied for a MyISAM or ARCHIVE table from one MySQL server to another on a different machine, and the second server accesses the table.

## **mysqldump**

- Dumps table contents to files:
  - All databases, specific databases, or specific tables
  - Allows back up local or remote servers
  - Independent of the storage engine
  - Written in text format
  - Portable
  - Excellent copy/move strategy
  - Good for small exports but not for full backup solution
- Basic usage:

```
mysqldump --user=<user> --password=<password>
--opt db_name > backup.file
```

### **Storage Location**

For SQL-format dump files that contain `CREATE TABLE` and `INSERT` statements for re-creating the tables, the server sends the table contents to `mysqldump`, which writes the files on the client host.

## Consistency with mysqldump

Ensuring consistency:

- **--master-data** option alone
  - Locks tables during backup
  - Records the binlog position in the backup file
- **--master-data** and **--single-transaction** options used together
  - Tables are not locked; only InnoDB table consistency is guaranteed.
- **--lock-all-tables**
  - Satisfies consistency by locking tables
- **--flush-logs**
  - Starts a new binary log

# mysqldump Output Format Options

- Drop options:
  - **--add-drop-database**
  - **--add-drop-table**
- Create options:
  - **--no-create-db**
  - **--no-create-info**
  - **--no-data**
  - **--no-tablespaces**
  - **--quick**
- MySQL programming components:
  - **--routines**
  - **--triggers**
- Top options in one option (**--opt**)

## Drop Options

`--add-drop-database` adds a `DROP DATABASE` statement before each `CREATE DATABASE` statement, while `--add-drop-table` adds a `DROP TABLE` statement before each `CREATE TABLE` statement.

## Create Options

`--no-create-db` suppresses the `CREATE DATABASE` statements, while `--no-create-info` suppresses the `CREATE TABLE` statements. `--no-data` creates the database and table structures but does not dump the data. `--no-tablespaces` tells the MySQL server not to write any `CREATE LOGFILE GROUP` or `CREATE TABLESPACE` statements to the output. `--quick` retrieves single rows from a table, without buffering sets of rows.

## MySQL Programming Components

`--routines` dumps stored routines (procedures and functions) from the dumped databases, while `--triggers` dumps triggers for each dumped table.

**The `--opt` option:** This is shorthand for the most common options to create an efficient and complete backup file.

# Restoring mysqldump Backups

- Reload **mysqldump** backups with **mysql**:

```
mysql --login-path=<login-path>
      <database> < backup_file.sql
```

- Name the database if the backup file does not.

- Copy from one database to another:

```
mysqldump -u<user> -p<password>
          <orig-db> <table> |
          mysql --login-path=<login-path> <copy-db>
```

- **mysqlimport**

- If **mysqldump** is invoked with the **--tab** option, it produces tab-delimited data files:
  - SQL file containing **CREATE TABLE** statements
  - Text file containing table data

## Reload mysqldump Backups

When using the **mysql** command for reloading **mysqldump** output, you must provide a name for the restored database if the dump file itself does not provide the name. If you create a dump file by invoking **mysqldump** with the **--database** or **--all-databases** option, you do not need to specify the target database name when reloading from that dump file. In that case, the dump file contains appropriate **USE db\_name** statements.

## Copy from One Database to Another

You can use **mysqldump** output to restore tables or databases and to copy them. **mysql** can read from a pipe, so the use of **mysqldump** and **mysql** can be combined into a single command that copies tables from one database to another. The pipe technique also can be used to copy databases or tables over the network to another server:

```
shell> mysqldump -u<user> -p<password> <orig-db> <table> |
        mysql --login-path=<login-path> <copy-db>
```

## **mysqlimport**

To reload the table, change location to the backup directory, process the `.sql` file by using `mysql`, and load the `.tsv` file by using `mysqlimport`:

```
shell> cd <backup_dir>
shell> mysql --login-path=<login-path> <database> < table.sql
shell> mysqlimport -u<user> -p<password> <database> table.tsv
```

If you combine the `--tab` option with options such as `--fields-terminated-by` and `--fields-enclosed-by`, which perform format control, specify the same format-control options with `mysqlimport` so that it knows how to interpret the data files.

## Privileges Required for `mysqldump`

You must have the following privileges to use `mysqldump`:

- `SELECT` for dumped tables
- `SHOW VIEW` for dumped views
- `TRIGGER` for dumped triggers
- `LOCK TABLES` (unless you use the `--single-transaction` option)
- Other options might require extra privileges.

To reload a dump file, you must have the following privileges:

- `CREATE` on each of the dumped objects
- `ALTER` on the database, if the `mysqldump` output includes statements that change the database collation to preserve character encodings

Some options require additional privileges. For example, you must have the `RELOAD` privilege to use the `--flush-logs` or `--master-data` options. Similarly, if you create a tab-delimited output with the `--tab` option, you must have the `FILE` privilege. If you want to back up stored functions and procedures by using the `--routines` option, you must have the `SELECT` privilege on the `mysql.proc` table.

## Quiz

**mysqldump** is good for small exports but not for a full backup solution.

- a. True
- b. False

# Backing Up Log and Status Files

- Binary log files
- Option files used by the server (**my.cnf** and **my.ini** files)
- Replication files
  - **master.info**
  - **relay-log.info**
- Replication slave data files
  - **SQL\_LOAD-\***
- MySQL binaries and libraries
- Strategies:
  - Static files: Backed up with normal system tools with the server running
  - Dynamic files: Backed up with normal system tools with the server stopped

## Binary Log Files

Binary logs store updates that have been made after the backup was made.

## Option Files Used by the Server (**my.cnf** and **my.ini** files)

These files contain configuration information that must be restored after a crash.

## Replication Files

Replication slave servers create a `master.info` file that contains information needed for connecting to the master server, and a `relay-log.info` file that indicates the current progress in processing the relay logs.

## Replication Slave Data Files

Replication slaves create data files for processing `LOAD DATA INFILE` statements. These files are located in the directory named by the `slave_load_tmpdir` system variable, which you can set by starting the server with the `--slave-load-tmpdir` option. If you do not set `slave_load_tmpdir`, the value of the `tmpdir` system variable applies. To safeguard replication slave data, back up the files beginning with `SQL_LOAD-`.

# Replication as an Aid to Backup

- The master can continue to run.
- The slave can be stopped to make a backup:
  - Stop the server.
  - `STOP SLAVE SQL_THREAD`
  - Flush tables.
- Back up the slave's databases:
  - Stopped servers can use system tools.
  - The slave thread is stopped, but the server that is still running can use any MySQL tools.
- Start the server:
  - Start server if it is stopped.
  - `START SLAVE SQL_THREAD`

If the MySQL server acts as a master in a replication setup, you can use a slave server to make backups instead of backing up the master. By using a slave server for backups, the master is not interrupted and the backup procedure does not add processing load on the master or require additional hard disk space or processing.

## Stopping the Slave

Shut down the `mysqld` process, or issue a `STOP SLAVE SQL_THREAD` statement to stop the server from processing updates that it receives from the master. In the latter case, you must flush the tables to force pending changes to disk.

## Back Up the Slave's Databases

Make a backup of the slave's databases. The allowable methods depend on whether the server is stopped or left running. For example, if the server is stopped, you cannot use tools such as `mysqldump` or `mysqlhotcopy` that connect to the server.

## Start Server

Restart the server if it was stopped. If the server was left running, you can restart the SQL thread by issuing a `START SLAVE SQL_THREAD` statement.

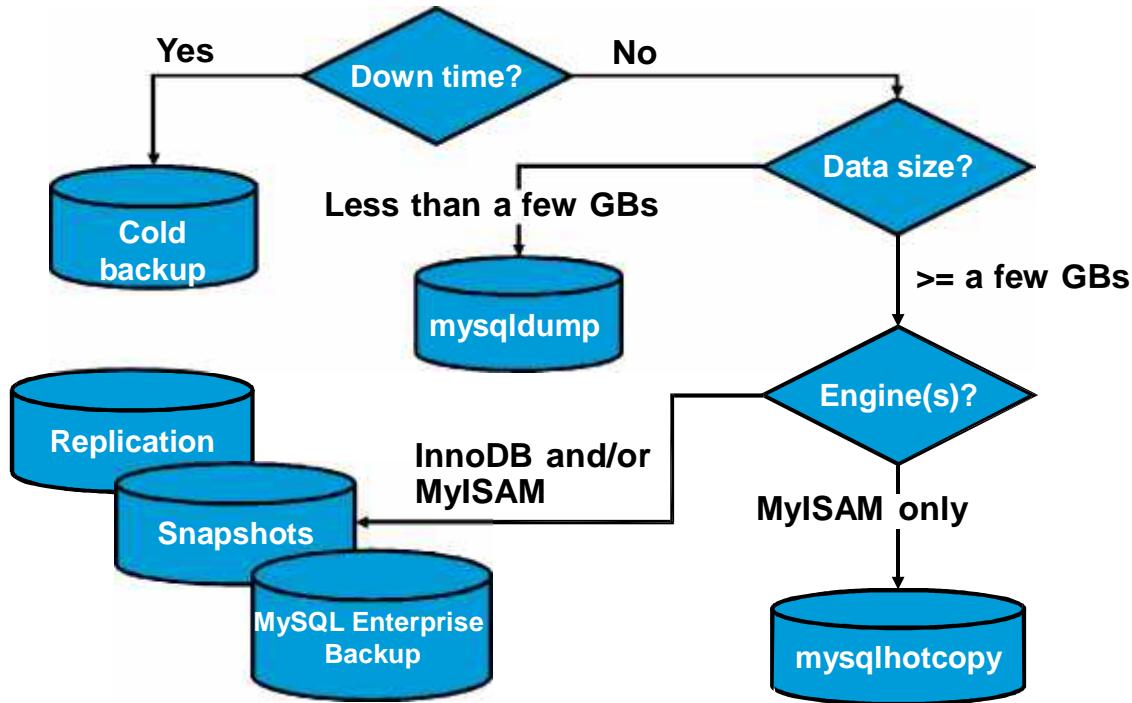
# Comparing Backup Methods

| Method                    | Hot/Warm/<br>Cold             | Storage<br>Engines | Logical/<br>Physical | Consistent | Availability                        |
|---------------------------|-------------------------------|--------------------|----------------------|------------|-------------------------------------|
| MySQL Enterprise Backup   | Hot (InnoDB)/<br>warm (other) | All                | Physical             | Yes        | Commercially available              |
| <code>mysqlhotcopy</code> | Warm                          | MyISAM             | Physical             | Yes        | Freely available                    |
| <code>mysqldump</code>    | Hot (InnoDB)/<br>warm (other) | All                | Logical              | Yes        | Freely available                    |
| Snapshots                 | Hot                           | All                | Physical             | Yes        | Need snapshot volume or file system |
| Replication               | Hot                           | All                | Logical or physical  | Yes        | Freely available                    |
| SQL Statements            | Warm                          | All                | Logical              | No         | Freely available                    |
| OS Copy Commands          | Cold or warm                  | All                | Physical             | Yes        | Freely available                    |

## Snapshots

Snapshots do not work in the same way for all engines. For example, InnoDB tables do not require `FLUSH TABLES WITH READ LOCK` to start a snapshot, but MyISAM tables do.

# Backup Strategy



The flowchart in the slide represents a decision-making process that you can use to determine a backup strategy. The questions you can ask in this process include:

- Can our system afford to be down for any length of time (down time)?
- How much data is there to back up?
- Which storage engines are being used to store the data (InnoDB, MyISAM, or both)?

## Processing Binary Log Contents

- Determine which logs were written after a backup was made.
- Convert the contents with `mysqlbinlog`:

```
mysqlbinlog bin.000050 bin.000051  
bin.000052 | mysql
```

- Process all binlogs with one command.

Restoring partial binlogs:

- `--start-datetime` / `--stop-datetime`
- `--start-position` / `--stop-position`

```
mysqlbinlog --start-position=23456  
binlog.000004 | mysql
```

After you restore the binary backup files or reload the text backup files, finish the recovery operations by reprocessing the data changes that are recorded in the server's binary logs. To do this, you must determine which logs were written after the backup was made. The contents of these binary logs then need to be converted to text SQL statements with the `mysqlbinlog` program to process the resulting statements with `mysql`.

## Point-in-Time Recovery

If a given binary log file was in the middle of being written during backup, you must extract from it only the part that was written after the backup, plus all log files written after that. Also, if a table or database is dropped by accident (or if other data corruption occurs), restore the backup by using the incremental activity recorded in the binary logs. To avoid re-executing the problem statement, you can run everything up to that statement by capturing the binary log file only up to that point. To handle partial-file extraction, `mysqlbinlog` supports options that enable the time to be specified or the log position at which to begin extracting log contents:

- `--start-datetime` option: Specifies the date and time at which to begin extraction, where the option argument is given in DATETIME format. Note that the granularity of `--start-datetime` is only one second, so it might not be accurate enough to specify the exact position to start with.
- `--start-position` option: Can be used to specify extraction beginning at a given log position
- There are also corresponding `--stop-datetime` and `--stop-position` options for specifying the point at which to stop extracting log contents.

If you are not sure about the time stamp or position in a log file that corresponds to the point at which processing begins, use `mysqlbinlog` without `mysql` to display the log contents for examination:

```
shell> mysqlbinlog file_name | more
```

## **Quiz**

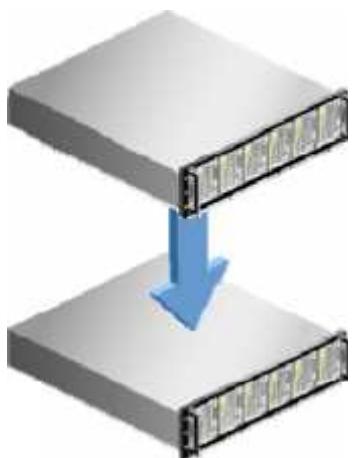
Which command converts the contents of binary logs?

- a. **binconvert**
- b. **mysql**
- c. **mysqlbinlog**
- d. **mysqldump**

## Chapter 11

### Introduction to Replication

# MySQL Replication



Replication is a feature of MySQL that allows servers to copy changes from one instance to another.

- The *master* records all data and structural changes to the binary log.
- The *slave* requests the binary log from the master and applies its contents locally.

## MySQL Replication

Replication in MySQL copies changes from one server—the master—to one or more slaves. The master writes changes to the binary log, and slaves request the master's binary log and apply its contents. The format of the log file affects how slaves apply changes. MySQL supports statement-based, row-based, and mixed format logging as described in the “Binary Logging Formats” slide in the lesson titled “Server Configuration.”

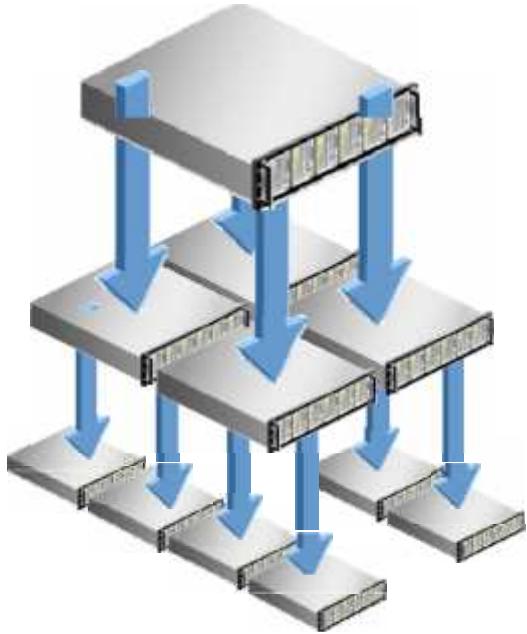
## Number of Slaves

There is no limit on how many slaves a single master can have. However, each additional slave uses a small amount of resources on the master, so you should carefully consider the benefit of each slave in production setups. The optimal number of slaves for a master in a given environment depends on a number of factors: size of schema, number of writes, relative performance of master and slaves, and factors such as CPU and memory availability. A general guideline is to limit the number of slaves per master to no more than 30.

## Network Outages

Replication in MySQL survives a network outage. Each slave keeps track of how much of the log it has processed, and resumes processing automatically when network connectivity is restored. This behavior is automatic and requires no special configuration.

# Replication Masters and Slaves



The master/slave relationship is *one-to-many*:

- Each slave reads logs from *one* master.
- A master can ship logs to *many* slaves.
- A slave can act as the master to another slave.

## Relay Slave

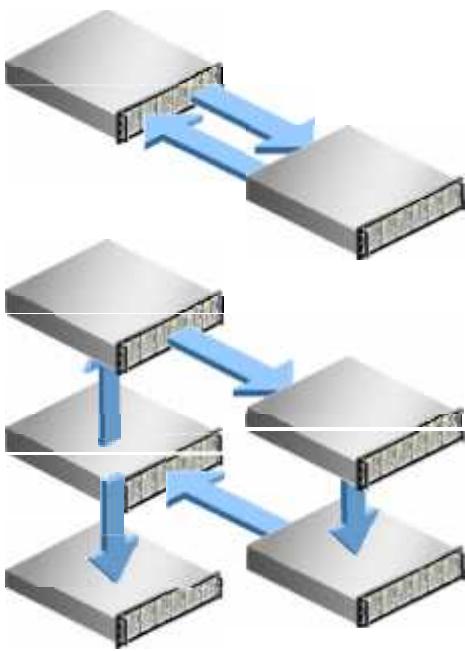
A slave can act as a master to another slave. Changes that occur at the top-most master are requested and applied by its immediate slaves, which relay the changes down to their slaves, and so on until replication reaches the end of the chain. This enables updates to propagate through multiple levels of replication, allowing for topologies that are more complex. Each additional level adds more propagation delays into the system, so a shallower setup suffers from less replication lag than a deeper one.

Each slave server can have only one master server; one slave cannot replicate from multiple master servers. A slave that acts as a master to other servers is often called a *relay slave*.

## Replicating with the BLACKHOLE Storage Engine

The BLACKHOLE storage engine silently discards all data changes with no warnings. The binary log continues to record those changes successfully. Use BLACKHOLE for specific tables on a relay slave when it replicates all changes to further slaves, but does not itself need to store data in those tables. For example, if you have a relay slave that is used solely for executing frequent long-running business intelligence reports on a small number of tables, you can configure all other replicated tables to use BLACKHOLE so that the server does not store data it does not need, while it replicates all changes to its slaves.

# Complex Topologies



More complex topologies are possible:

- A *bi-directional topology* has two master servers, each a slave of the other.
- A *circular topology* has any number of servers.
  - Each is both a master and a slave of another master.
  - Changes on any master replicate to all masters.
- Not every slave must be a master.

**Note:** MySQL replication does not perform conflict resolution.

## Conflict Resolution

In a typical configuration, clients write changes only to the master but read changes from any server. In an environment where servers allow concurrent updates on similar data, the eventual state of the data on multiple servers might become inconsistent. It is the responsibility of the application to prevent or manage conflicting operations. MySQL replication does not perform conflict resolution.

Conflicts can occur in any topology that includes more than one master. This includes simple hierarchies such as that shown in the preceding slide, if a relay slave accepts changes from clients. Conflicts are particularly prevalent in circular topologies.

For example, consider an e-commerce company that implements replication using a circular topology, in which two of the servers deal with applications in the “Luxury Brands” and “Special Events” teams, respectively. Assume that the applications do not manage conflicting operations, and the following events occur:

- The “Luxury Brands” team increases the price of luxury products by 20%.
- The “Special Events” team reduces the prices of all products over \$500 by \$50 because of an upcoming special holiday.
- One product costing \$520 falls into both categories and has its value updated by both of the preceding operations.

The eventual price of the product on each server depends on the order in which each server performs the operations. If both operations occur almost at the same time, the following operations occur:

- The server in the “Luxury Brands” team increases the product’s price by 20%, from \$520 to \$624
- The server in the “Special Events” team reduces the product’s price by \$50, from \$520 to \$470
- The changes each replicate to the other server, resulting in the “Luxury Brands” server assuming a final value for that product of \$574, and the “Special Events” server assuming a final value of \$564.
- Other servers in the replication environment assume final values depending on the order in which they applied the operations.

Similarly, conflicts occur if the “Luxury Brands” team adds a new product that has not replicated fully by the time the “Special Events” team makes its changes, or if two teams add a product with the same primary key on different servers. While row-based replication solves some conflicts, many conflicts can be prevented only at the application level.

**Note:** MySQL Cluster uses a form of replication internally that differs in some ways from replication in MySQL Server, and offers conflict detection (and optional resolution).

# Replication Use Cases

Common uses for replication:

- **Horizontal scale-out:** Spread the querying workload across multiple slaves.
- **Business intelligence and analytics:** Run expensive reports and analytics on a slave, letting the master focus on production applications.
- **Geographic data distribution:** Serve local users with a local application, and replicate business intelligence data to corporate servers.

## Common Use Cases

- **Horizontal scale-out:** The most popular reason to implement replication is to spread the querying workload across one or more slave servers to improve the performance of read operations throughout the application, and write operations on the master server by reducing its read workload. See <http://dev.mysql.com/doc/mysql/en/faqs-replication.html#qandaitem-B-13-1-8> for a worked example of how you can improve the performance of your system by using replication for scale-out.
- **Business intelligence and analytics:** Business intelligence reports and analytical processing can be resource intensive and can take considerable time to execute. In a replicated environment, you can run such queries on a slave so that the master can continue to service the production workload without being affected by long-running and I/O intensive reports.
- **Geographic data distribution:** Companies with a distributed geographic presence can benefit from replication by having servers in each region that process local data and replicate that data across the organization. This provides the performance and administrative benefits of geographic proximity to customers and the workforce, while also enabling corporate awareness of data throughout the company.

**Note:** Multisource replication is possible only indirectly with circular topologies.

## Replication for High Availability

Replication allows various high-availability use cases.

- **Controlled switchover:** Use a replica to act in place of a production server during hardware or system upgrades.
- **Server Redundancy:** Perform a failover to a replica server in case of a system failure.
- **Online schema changes:** Perform a rolling upgrade in an environment with several servers to avoid an overall system outage.
- **Software upgrades:** Replicate across different versions of MySQL during an environment upgrade.
  - Slaves must run a later version than the master.
  - Queries issued during the upgrade must be supported by all versions used during the upgrade process.

Replication allows you to fail over to the slave server if the master goes offline due to hardware or software failure, or to perform scheduled maintenance.

# Configuring Replication

- Configure a unique **server-id** for each server.
- Configure each master:
  - Enable the binary log, and enable TCP/IP networking.
  - Create a new user with the **REPLICATION SLAVE** privilege.
  - Back up the master databases, and record the log coordinates if required.
- Configure each slave
  - Restore the backup from the master.
  - Issue a **CHANGE MASTER TO** statement on each slave with the:
    - Network location of the master
    - Replication account username and password
    - Log coordinates from which to start replicating if required
  - Start replication with **START SLAVE**.

Every server in a replication topology must have a unique `server-id`, an unsigned 32-bit integer with a value from 0 (the default) to 4,294,967,295. A server with a `server-id` of 0, whether slave or master, refuses to replicate with other servers.

## Master Configuration

Every master must have an assigned IP address and TCP port, because replication cannot use UNIX socket files. Each master must also enable binary logging, because during replication, each master sends its log contents to each slave.

Each slave must log on to a master to replicate from it. If you are creating a new user on the master for that purpose, that new user must have the **REPLICATION SLAVE** privilege:

```
GRANT REPLICATION SLAVE ON *.* TO <user>@<slave-hostname>;
```

If you are creating a replication topology using a master that already contains a populated database, you must create a copy of that database as a starting point for the slave (for example, by performing a backup of the master and restoring that backup to the slave).

If you are using Global Transaction Identifiers, you do not need to record log coordinates.

## Slave Configuration

Each slave connects to only one master. To tell the slave about the master, use the **CHANGE MASTER TO...** statement.

## CHANGE MASTER TO

- Issue the CHANGE MASTER TO... statement on the slave to configure replication master connection details:

```
mysql> CHANGE MASTER TO
      ->   MASTER_HOST = 'host_name',
      ->   MASTER_PORT = port_num,
      ->   MASTER_USER = 'user_name',
      ->   MASTER_PASSWORD = 'password',
      ->   MASTER_LOG_FILE = 'master_log_name',
      ->   MASTER_LOG_POS = master_log_pos;
```

- Subsequent invocations of CHANGE MASTER TO retain the value of each unspecified option.
  - Changing the master's host or port also resets the log coordinates.
  - The following statement changes the password, but retains all other settings:

```
mysql> CHANGE MASTER TO MASTER_PASSWORD='newpass';
```

The MASTER\_HOST and MASTER\_PORT values specify the hostname and TCP port number of the master. The MASTER\_USER and MASTER\_PASSWORD values specify the account details of an account on the master with the REPLICATION SLAVE privilege. To improve security, you can also use MASTER\_SSL and related options on SSL-enabled servers to encrypt network traffic between slave and master during replication.

The MASTER\_LOG\_FILE and MASTER\_LOG\_POS values contain the log coordinates of the binary log position from which the slave starts replicating. You can get the file and position from the master by executing the SHOW MASTER STATUS statement:

```
mysql> SHOW MASTER STATUS;
+-----+-----+-----+-----+
| File      | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-----+-----+-----+-----+
| mysql-bin.000014 |    51467 |           |           |           |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

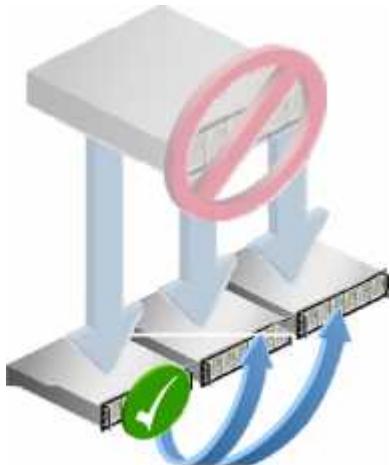
If you use mysqldump to perform a backup of the master's databases as a starting point for the slave, you can use the --master-data option to include log coordinates in the backup:

```
mysqldump -uroot -p --master-data -B world_innodb > backup.sql
```

Specify MASTER\_AUTO\_POSITION=1 instead of log coordinates if you are using GTIDs.

## Failover with Log Coordinates

- To find the new master and the correct log coordinates for each slave, you must examine the binary logs closely.



- Find the most recent event applied to each slave.
- Select an up-to-date slave as a new master.
- Identify the log coordinates on the new master to match the latest applied event on each other slave.
- Issue the correct `CHANGE MASTER TO...` on each slave.

- In circular topologies, finding the source of events in each binary log becomes very difficult.

To fail over after a master becomes unavailable, stop all slaves and select a slave as the new master by issuing a `CHANGE MASTER TO` statement (with the log coordinates of that new master) on the remaining slaves. If the slaves are not all up-to-date when you failover, you risk having an inconsistent replication topology:

- If the new master is behind a particular slave (that is, if the slave has already applied the events at the end of the new master's log), the slave repeats those events.
- If the new master is ahead of a particular slave (that is, if the new master's binary log contains events that the slave has not yet applied), the slave skips those events.

To avoid such inconsistencies, you must select an up-to-date slave as new master, and then find the log coordinates on the new master that match the most recent event on each slave. If some slaves are more behind than others, the log coordinates that you provide in the `CHANGE MASTER TO...` statement differ from one slave to the next, so you cannot simply issue a `SHOW MASTER STATUS` on the new master. Instead, you must examine the binary logs to find the correct coordinates.

In circular topologies with multiple masters accepting client updates, finding an up-to-date slave and identifying correct log coordinates can be very difficult, because each slave applies operations in a different order to other slaves. To avoid this difficulty, use Global Transaction Identifiers (GTIDs). The MySQL Utilities also include tools to facilitate failover with GTIDs.

## Global Transaction Identifiers (GTIDs)

Global Transaction Identifiers (GTIDs) uniquely identify each transaction in a replicated network.

- Each GTID is of the form <source-uuid>:<transaction-id>.   
**0ed18583-47fd-11e2-92f3-0019b944b7f7:338**
- A GTID set contains a range of GTIDs:   
**0ed18583-47fd-11e2-92f3-0019b944b7f7:1-338**
- Enable GTID mode with the following options:
  - **gtid-mode=ON**: Logs a unique GTID along with each transaction
  - **enforce-gtid-consistency**: Disallows events that cannot be logged in a transactionally safe way
  - **log-slave-updates**: Records replicated events to the slave's binary log

The source UUID (universally unique identifier) is the UUID of the originating server for each transaction. Each server's UUID is stored in the `auto.cnf` file in the data directory. If that file does not exist, MySQL creates the file and generates a new UUID, placing it in the new file.

Query a server's UUID with the `server_uuid` variable:

```
mysql> SELECT @@server_uuid\G
***** 1. row *****
@@server_uuid: 0ed18583-47fd-11e2-92f3-0019b944b7f7
1 row in set (0.00 sec)
```

When a client executes a transaction on a master, MySQL creates a new GTID and logs the transaction with its unique GTID. When a slave reads that transaction from the master and applies it, the transaction retains its original GTID. That is, the server UUID of a transaction replicated to a slave is that of the master, not the slave. Subsequent slaves in a replication chain each record the transaction along with its original GTID. As a result, every slave in a replication topology can identify the master on which a transaction first executed.

When each server logs a transaction, it also records that transaction's ID in a GTID set within the `gtid_executed` variable. In a global context, this variable contains all GTID sets (representing all transactions from this and other upstream master servers) logged to the server's binary log.

```
mysql> SELECT @@global.gtid_executed\G
***** 1. row *****
@@global.gtid_executed: bacc034d-4785-11e2-8fe9-0019b944b7f7:1-34,
c237b5cd-4785-11e2-8fe9-0019b944b7f7:1-9,
c9cec614-4785-11e2-8fea-0019b944b7f7:1-839
1 row in set (0.00 sec)
```

The `gtid_purged` variable contains GTID sets that have been purged from the binary log. Both `gtid_purged` and the global `gtid_executed` are reset to an empty string when you execute `RESET MASTER` on the server.

## Replication with GTIDs

Use `CHANGE MASTER TO...` to enable GTID replication:

- Tell the slave that transactions are identified by GTIDs:

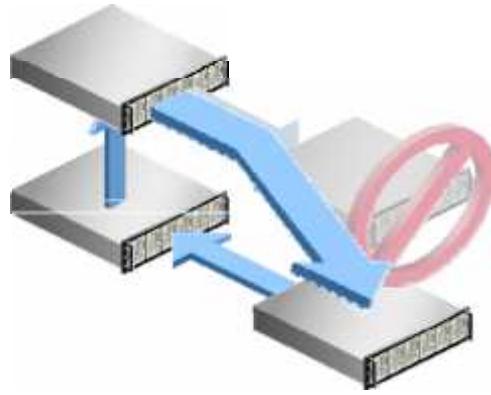
```
CHANGE MASTER TO MASTER_AUTO_POSITION=1;
```

- You do not need to provide log coordinates such as:
  - `MASTER_LOG_FILE`
  - `MASTER_LOG_POS`
- You cannot provide `MASTER_AUTO_POSITION` and log coordinates in the same `CHANGE MASTER TO...` statement.

When you enable GTID-based replication, you do not need to specify the log coordinates of the master, because the slave sends the value of `@@global.gtid_executed` to the master. Therefore, the master knows which transactions the slave has executed, and sends only those transactions that the slave has not already executed.

## Failover with GTIDs

- When you use GTIDs, failover in a circular topology is trivial.
  - On the slave of the failed master, bypass it by issuing a single `CHANGE MASTER TO` statement.
  - Each server ignores or applies transactions replicated from other servers in the topology, depending on whether the transaction's GTID has already been seen or not.
- Failover in a non-circular topology is similarly easy.
  - Temporarily configure the new master as a slave of an up-to-date slave until it has caught up.



Although GTIDs prevent the duplication of events that originated on a single server, they do not prevent conflicting operations that originated on different servers, as described in the slide titled “Complex Topologies.” When reconnecting applications to your servers following a failover, you must be careful not to introduce such conflicts.

# Replication Filtering Rules

Filters are server options that apply to a *master* or *slave*:

- The master applies **binlog-\*** filters when writing the binary log.
- The slave applies **replicate-\*** filters when reading the relay log.

Choose which events to replicate, based on:

- Database:
  - **replicate-do-db**, **binlog-do-db**
  - **replicate-ignore-db**, **binlog-ignore-db**
- Table:
  - **replicate-do-table**, **replicate-wild-do-table**
  - **replicate-ignore-table**,  
**replicate-wild-ignore-table**

Use filtering rules when different slaves in an environment serve different purposes. For example, a server dedicated to displaying web content does not need to replicate restocking information or payroll records from the master, while a server dedicated to generating management reports on sales volume does not need to store web content or marketing copy.

Filtering rules have complex precedence rules that apply in order:

- Database filters apply before table filters.
- Table wildcard filters `*-wild-*` apply after those that do not use wildcards.
- The `*-do-*` filters apply before the respective `*-ignore-*` filters.

Be careful when using multiple filters. It is very easy to make mistakes, due to the complex order in which filters are applied. Because filters control what data is replicated, such mistakes are difficult to recover from. For this reason, do not mix different types of filters. For example, if you use `replicate-wild-*`, do not use any non-wild `replicate-*` filters.

To suppress replication of all temporary tables, set `binlog_format=ROW`. To suppress replication of a specific temporary table, use `replicate-ignore-table`. See <http://dev.mysql.com/doc/mysql/en/replication-features-tempTables.html> for further information. For a full discussion of replication rules and their execution order, see:

<http://dev.mysql.com/doc/mysql/en/replication-rules.html>.

## **MySQL Utilities**

MySQL Utilities are command-line tools that provide a number of useful features. They are:

- Distributed with MySQL Workbench
- Used for maintenance and administration of MySQL servers
- Written in Python, and easily extensible by Python programmers using a supplied library
- Useful for configuring replication topologies and performing failover

# MySQL Utilities for Replication

Several utilities are particularly useful for replication:

- **mysqldbcopy**: Copies a database from a source server to a destination server along with replication configuration
- **mysqldbcompare**: Compares two databases to find differences and create a script to synchronize them
- **mysqlrpladmin**: Administers a replication topology
  - Failover to the best slave after a master failure
  - Switchover to promote a specified slave
  - Start, reset, or stop all slaves
- **mysqlfailover**: Monitors a master continuously, and executes failover to the best slave available

The `mysqldbcopy` utility accepts the option `--rpl` that runs a `CHANGE MASTER TO` statement on the destination server. It accepts the following values:

- **master**: The destination server becomes a slave of the source.
- **slave**: The source is already a slave of another master. The destination server copies that master information from the source and becomes a slave of the same master

The `mysqlfailover` utility checks the master status periodically with a ping operation. Both the period interval and ping operation are configurable. It uses GTIDs to ensure that a new master is fully up-to-date when it becomes the master, doing so by choosing a new master from a list of candidates (optionally from all slaves if no candidate in the list is viable), configuring it as a slave of all other slaves to collect all outstanding transactions, before finally making it the new master. The `failover` command of `mysqlrpladmin` performs a similar temporary reconfiguration when selecting a new master.

You can also perform health monitoring without performing automatic reconfiguration if a master fails.

# MySQL Utilities for Replication

- **mysqlrplcheck**: Checks the prerequisites for replication between a master and a slave, including:
  - Binary logging
  - Replication user with appropriate privileges
  - server\_id conflicts
  - Various settings that can cause replication conflicts
- **mysqlreplicate**: Starts replication between two servers, reporting warnings for mismatches
- **mysqlrplshow**: Displays a replication topology between a master and slave or recursively for the whole topology

The mysqlrplshow utility displays a replication topology such as the following:

```
# Replication Topology Graph
localhost:3311 (MASTER)
|
+--- localhost:3312 - (SLAVE + MASTER)
|
+--- localhost:3313 - (SLAVE + MASTER)
|
+--- localhost:3311 <--> (SLAVE)
```

In the preceding example, the server at port 3311 appears twice: once as master and once as slave. The <--> symbol indicates circularity within the topology.

# Asynchronous Replication

- The slave requests the binary log and applies its contents.
  - The slave typically lags behind the master.
- The master does not care when the slave applies the log.
  - The master continues operating without waiting for the slave.

During MySQL replication in its default configuration, the master server accepts change events from clients, committing those events and writing them to the binary log. In a separate thread, the master streams the binary log to connected slaves. Because the master commits changes without waiting for a response from any slave, this is called *asynchronous* replication.

Most importantly, this means that slaves have not yet applied transactions at the time the master reports success to the application. Usually, this is not a problem. However, if the master crashes with data loss after committing a transaction but before the transaction replicates to any slave, the transaction is lost even though the application has reported success to the user.

If the master waited for all slaves to apply its changes before committing the transaction, replication would then be called *synchronous*. Although MySQL replication is not synchronous, MySQL Cluster uses synchronous replication internally to ensure data consistency throughout the cluster, and MySQL client requests are synchronous because the client waits for a server response after issuing a query to the server.

# Semisynchronous Replication

- Semisynchronous replication:
  - Requires a plugin on the master and at least one slave
  - Blocks each master event until at least one slave receives it
  - Switches to asynchronous replication if a timeout occurs

To enable semisynchronous replication, install a plugin on the master and on at least one slave. Use the following plugins:

- `rpl_semi_sync_master` on a master
- `rpl_semi_sync_slave` on a slave

You must also enable the following options:

- `rpl_semi_sync_master_enabled` on a master
- `rpl_semi_sync_slave_enabled` on a slave

If you enable semisynchronous replication on a master, it behaves asynchronously until at least one semisynchronous slave connects.

During semisynchronous replication, the master blocks after committing a transaction until at least one semisynchronous slave acknowledges that it too has received the transaction. This means that at least one slave has received each transaction at the time the master reports success to the application. If the master crashes with data loss after committing a transaction and the application has reported success to the user, the transaction also exists on at least one slave.

Semisynchronous replication presents you with a trade-off between performance and data integrity. Transactions are slower when using semisynchronous replication than when using asynchronous replication because the master waits for a slave response before committing.

The extra time taken by each transaction is at least the time it takes for:

- The TCP/IP roundtrip to send the commit to the slave
- The slave recording the commit in its relay log
- The master waiting for the slave's acknowledgment of that commit

This means that semisynchronous replication works most effectively for servers that are physically co-located, communicating over fast networks.

If the master does not receive a response from a semisynchronous slave within a timeout period, the master still commits the transaction but reverts to asynchronous mode. You can configure the timeout with the `rpl_semi_sync_master_timeout` variable, which contains a value measured in milliseconds. The default value is 10000, representing ten seconds.

## **Quiz**

To use Global Transaction Identifiers in replication, you must use a CHANGE MASTER TO... statement to provide the binary log coordinates (file name and position) of the most recent transaction on the master that the slave has not yet applied.

- a. True
- b. False

# Review of Binary Logging

The binary log:

- Contains data and schema changes, and their time stamps
  - *Statement-based* or *row-based* logging
- Is used for point-in-time recovery from backup, full recovery from backup, and replication
- Rotates when:
  - MySQL restarts
  - It reaches its maximum size as set by `max_binlog_size`
  - You issue a `FLUSH LOGS` statement
- Can be inspected in various ways:
  - Metadata: `SHOW BINARY LOGS`, `SHOW MASTER STATUS`
  - Contents: `mysqlbinlog`

The material in this slide is a summary of content covered in the lesson titled “Server Configuration.”

## **FLUSH LOGS in Replication Environments**

By default, the MySQL server writes `FLUSH` commands to the binary log so that statements are replicated to the slaves. To prevent this from happening, use the optional `NO_WRITE_TO_BINLOG` keyword or its alias `LOCAL`.

# Replication Logs

Slave servers maintain information about replication events.

- Relay log set:
  - Includes relay logs and relay log index file
  - Contains a copy of binary log events from the master
- Slave status logs:
  - Contain information needed to perform replication
    - Master connection details and log coordinates
    - Relay log coordinates
  - Are stored in files or tables
    - `master.info` and `relay-log.info` files by default
    - `slave_master_info` and `slave_relay_log_info` tables in the `mysql` database

**Relay logs:** MySQL automatically manages the set of relay log files, removing files when it has replayed all events and creating a new file when the current file exceeds the maximum relay log file size (or `max_binlog_size` if `max_relay_log_size` is 0). The relay logs are stored in the same format as the binary logs; you can view them with `mysqlbinlog`. The slave maintains an index file to keep track of relay log files.

The relay log files are named `<host_name>-relay-bin.<nnnnnnn>` by default, and the index file is named `<host_name>-relay-bin.index`. To make the server configuration immune to potential future hostname changes, change these host name prefixes by setting the options `--relay-log` and `--relay-log-index`.

**Slave status logs:** The slave stores information about how to connect to the master, and the most recently replicated log coordinates of the master's binary log and the slave's relay log.

There are two such logs:

- **Master information:** This log contains information about the master server, including information such as the host name and port, credentials used to connect, and the most recently downloaded log coordinates of the master's binary log.
- **Relay log information:** This log contains the most recently executed coordinates of the relay log, and the number of seconds by which the slave's replicated events are behind those of the master.

# Crash-Safe Replication

- Binary logging is crash-safe:
  - MySQL only logs complete events or transactions.
  - Use `sync-binlog` to improve safety.
    - By default, the value is 0, meaning the operating system writes to the file according to its internal rules.
    - Set `sync-binlog` to 1 to force the operating system to write the file after every transaction, or set it to any larger number to write after that number of transactions.
- Store slave status logs in tables for crash-safe replication.
  - Options: `master-info-repository` and `relay-log-info-repository`
    - Possible values are `FILE` (the default) and `TABLE`.
    - `TABLE` is crash-safe.

## Crash-Safe Replication

The slave status logs are stored in files by default. Use the `master-info-file` and `relay-log-info-file` options to set the file names. By default, the file names are `master.info` and `relay-log.info`, both stored in the data directory.

If you store the slave status logs in files, a crash could happen at a point between logging an event and recording that event's log coordinates in the status log. When the server restarts after such an event, the status file and binary log are inconsistent, and recovery becomes difficult.

When replicating data that uses a transactional storage engine such as InnoDB, store the status logs in transactional tables to improve performance and ensure crash-safe replication by changing the values of `master-info-repository` and `relay-log-info-repository` from `FILE` (the default) to `TABLE`. The tables are called `slave_master_info` and `slave_relay_log_info`, both stored in the `mysql` database, and both using the InnoDB engine to ensure transactional integrity and crash-safe behavior.

For more information on the slave status logs, see: <http://dev.mysql.com/doc/mysql/en/slave-logs-status.html>

# Replication Threads

When a slave connects to a master:

- The master creates a *Binlog dump thread*
  - Reads events from the binary log and sends them to the slave I/O thread
- The slave creates at least two threads:
  - Slave I/O thread
    - Reads events from the master's Binlog dump thread and writes them to the slave's relay log
  - Slave SQL thread
    - Applies relay log events on single-threaded slave
    - Distributes relay log events between worker threads on multithreaded slave
  - Slave worker threads
    - Apply relay log events on multithreaded slave

MySQL creates threads on both the master and slave servers to perform the work of replication.

When a slave connects successfully to a master, the master server launches a replication master thread, known as the *Binlog dump thread*, displayed as “Binlog Dump GTID” if the slave is configured to use the auto-positioning protocol (`CHANGE MASTER TO MASTER_AUTO_POSITION`). This thread sends events within the binary log to the slave as those events arrive, while the slave is connected. The master creates a Binlog dump thread for each connected slave.

By default, each slave launches two threads, known as the *Slave I/O* thread and the *Slave SQL* thread, respectively.

- The Slave I/O thread connects to the master, and reads updates from the Binlog dump thread into a local relay log.
- The Slave SQL thread executes the events within the relay log.

The default configuration, called *single-threaded* due to the single SQL thread processing the relay log, can result in slave lag, where the slave falls behind the master: The master applies changes in parallel if it has multiple client connections, but it serializes all events in its binary log. The slave executes these events sequentially in a single thread, which can become a bottleneck in high volume environments, or when the slave's hardware is not powerful enough to deal with the volume of traffic in a single thread.

## Multithreaded Slaves

MySQL supports multithreaded slaves to avoid some of the lag caused by single-threaded slaves. If you set the `slave_parallel_workers` variable to a value greater than zero on a slave, it creates that many worker threads. In that case, the Slave SQL thread does not execute events directly. Instead, it distributes events to the worker threads on a per-database basis. This makes multithreaded slaves particularly useful in environments where you replicate data in multiple databases.

This option can cause inconsistency between databases if worker threads perform parallel operations in an order different from how they were performed on the master, so you must ensure that data in different databases is independent. Data within one database, being replicated by a single thread, is guaranteed to be consistent.

If you set the `slave_parallel_workers` variable to a value larger than the number of databases used in replication, some worker threads remain idle. Similarly, if you replicate only one database across your servers, there is no benefit to using multithreaded slaves in your environment.

# Controlling Slave Threads

- Control slave threads:

```
START SLAVE;  
STOP SLAVE;
```

- Control threads individually:

```
START SLAVE IO_THREAD;  
STOP SLAVE SQL_THREAD;
```

- Start threads until a specified condition:

```
START SLAVE UNTIL SQL_AFTER_MTS_GAPS;  
START SLAVE IO_THREAD  
UNTIL SQL_AFTER_GTIDS =  
0ed18583-47fd-11e2-92f3-0019b944b7f7:338;
```

- Disconnect the slave from the master:

```
RESET SLAVE [ALL]
```

You can start and stop the slave's SQL and I/O threads with the `START SLAVE` and `STOP SLAVE` statements. Without arguments, these statements control both threads. Control each thread individually by specifying `IO_THREAD` or `SQL_THREAD` as an argument. For example, you can temporarily stop the slave from querying the master by issuing the following statement on the slave:

```
STOP SLAVE IO_THREAD;
```

When starting a slave thread, you can choose to have the thread stop when it gets to a point specified in the `UNTIL` clause. You can specify this point using log coordinates, a GTID set, or the special `SQL_AFTER_MTS_GAPS` value, used with multithreaded slaves.

For example, the following statement starts the SQL thread on a slave until it executes the specified GTID from the relay log, at which point it stops:

```
START SLAVE SQL_THREAD UNTIL  
SQL_AFTER_GTIDS = 0ed18583-47fd-11e2-92f3-0019b944b7f7:338
```

**Note:** Slave worker threads cannot be started or stopped individually, because the SQL thread controls the work they do. Start or stop the SQL thread to control the worker threads.

For complete details on using `START SLAVE` see: <http://dev.mysql.com/doc/mysql/en/start-slave.html>

## Resetting the Slave

Sometimes you might want to disconnect the slave from the master for a clean start. The `RESET SLAVE` command does the following:

- Clears `master.info` and `relay.log` repositories
- Deletes all the relay logs
- Starts a new relay log file
- Resets any `MASTER_DELAY` specified in the `CHANGE MASTER TO` statement to 0
- Retains connection parameters (only in MySQL 5.6 onwards), so you can restart the slave without executing `CHANGE MASTER TO`.
  - Issue `RESET SLAVE ALL` to reset the connection parameters.

# Monitoring Replication

```
mysql> SHOW SLAVE STATUS\G
***** 1. row *****
Slave_IO_State: Queueing master event to the relay log
...
Master_Log_File: mysql-bin.005
Read_Master_Log_Pos: 79
Relay_Log_File: slave-relay-bin.005
Relay_Log_Pos: 548
Relay_Master_Log_File: mysql-bin.004
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
Exec_Master_Log_Pos: 3769
...
Seconds_Behind_Master: 8
```

**Slave\_\*\_Running:** The Slave\_IO\_Running and Slave\_SQL\_Running columns identify whether the slave's I/O thread and SQL thread are currently running, not running, or running but not yet connected to the master. The respective possible values are Yes, No, or Connecting.

**Master Log Coordinates:** The Master\_Log\_File and Read\_Master\_Log\_Pos columns identify the coordinates of the most recent event in the master's binary log that the I/O thread has transferred. These columns are comparable to the coordinates shown when you execute SHOW MASTER STATUS on the master. If the values of Master\_Log\_File and Read\_Master\_Log\_Pos are far behind those on the master, this indicates latency in the network transfer of events between the master and slave.

**Relay Log Coordinates:** The Relay\_Log\_File and Relay\_Log\_Pos columns identify the coordinates of the most recent event in the slave's relay log that the SQL thread has executed. These coordinates correspond to coordinates in the master's binary log identified by the Relay\_Master\_Log\_File and Exec\_Master\_Log\_Pos columns.

If the Relay\_Master\_Log\_File and Exec\_Master\_Log\_Pos columns' output is far behind the Master\_Log\_File and Read\_Master\_Log\_Pos columns representing the coordinates of the I/O thread, this indicates latency in the SQL thread rather than the I/O thread. That is, it indicates that copying the log events is faster than executing them.

On a multithreaded slave, `Exec_Master_Log_Pos` contains the position of the last point before any uncommitted transactions. This is not always the same as the most recent log position in the relay log, because a multithreaded slave might execute transactions on different databases in a different order from that represented in the binary log.

**Seconds\_Behind\_Master:** This column provides the number of seconds between the time stamp (on the master) of the most recent event in the relay log executed by the SQL thread and the real time of the slave machine. A delay of this type normally occurs when the master processes a large volume of events in parallel that the slave must process serially, or when the slave's hardware is not capable of handling the same volume of events that the master can handle during periods of high traffic. If the slave is not connected to the master, this column is `NULL`.

**Note:** This column does not show latency in the I/O thread or in the network transfer of events from the master.

## Replication Slave I/O Thread States

The most commonly seen I/O thread states are:

- Connecting to master
- Waiting for master to send event
- Queueing master event to the relay log
- Waiting to reconnect after a failed binlog dump request

The lists in this slide and the next slide show the most common states displayed in the State column of the output of `SHOW PROCESSLIST` for a slave server I/O thread. These states also appear in the `Slave_IO_State` column displayed by `SHOW SLAVE STATUS`.

- **Connecting to master:** The thread is attempting to connect to the master.
- **Waiting for master to send event:** The thread has connected to the master and is waiting for binary log events to arrive. This can last for a long time if the master is idle. If the wait lasts for `slave_read_timeout` seconds, a timeout occurs. At that point, the thread considers the connection to be broken and attempts to reconnect.
- **Queueing master event to the relay log:** The thread has read an event and is copying it to the relay log so that the SQL thread can process it.
- **Waiting to reconnect after a failed binlog dump request:** If the binary log dump request failed (due to disconnection), the thread goes into this state while it sleeps, and then tries to reconnect periodically. The interval between retries can be specified using the `--master-connect-retry` option.

## Replication Slave I/O Thread States

- Reconnecting after a failed binlog dump request
- Waiting to reconnect after a failed master event read
- Reconnecting after a failed master event read
- Waiting for the slave SQL thread to free enough relay log space

- **Reconnecting after a failed binlog dump request:** The thread is trying to reconnect to the master.
- **Waiting to reconnect after a failed master event read:** An error occurred while reading (due to disconnection). The thread is sleeping for `master-connect-retry` seconds before attempting to reconnect.
- **Reconnecting after a failed master event read:** The thread is trying to reconnect to the master. When connection is established again, the state becomes Waiting for master to send event.
- **Waiting for the slave SQL thread to free enough relay log space:** This state shows that a non-zero `relay_log_space_limit` value is being used, and the relay logs have grown large enough so that their combined size exceeds this value. The I/O thread is waiting until the SQL thread frees enough space by processing relay log contents so that it can delete some relay log files.

## Replication Slave SQL Thread States

The most commonly seen SQL thread states are:

- Waiting for the next event in relay log
- Reading event from the relay log
- Making temp file
- Slave has read all relay log; waiting for the slave I/O thread to update it
- Waiting until MASTER\_DELAY seconds after master executed event

The following state appears in worker threads:

- Waiting for an event from Coordinator

The slide shows the most common states displayed in the State column for slave SQL threads and worker threads.

- **Waiting for the next event in relay log:** This is the initial state before Reading event from the relay log.
- **Reading event from the relay log:** The thread has read an event from the relay log so that the event can be processed.
- **Making temp file:** The thread is executing a LOAD DATA INFILE statement and is creating a temporary file containing the data from which the slave reads rows.
- **Slave has read all relay log; waiting for the slave I/O thread to update it:** The thread has processed all events in the relay log files and is now waiting for the I/O thread to write new events to the relay log.
- **Waiting until MASTER\_DELAY seconds after master executed event:** The SQL thread has read an event but is waiting for the slave delay to lapse. This delay is set with the MASTER\_DELAY option of CHANGE MASTER TO.
- **Waiting for an event from Coordinator:** On multithreaded slaves, a worker is waiting for the coordinator thread to assign a job to the worker queue.

# Troubleshooting MySQL Replication

- View the error log.
  - The error log can provide you with enough information to identify and correct problems in replication.
- Issue a **SHOW MASTER STATUS** statement on the master.
  - Logging is enabled if the position value is non-zero.
- Verify that both the master and slave have a unique non-zero server ID value.
  - The master and the slave must have different server IDs.
- Issue a **SHOW SLAVE STATUS** command on the slave.
  - Slave\_IO\_Running and Slave\_SQL\_Running display Yes when the slave is functioning correctly.
  - Last\_IO\_Error and Last\_SQL\_Error show the most recent error messages from the I/O and SQL threads.

The **SHOW SLAVE STATUS** statement returns information in several fields related to the most recent error encountered during replication.

- **Last\_IO\_Error, Last\_SQL\_Error:** The error message of the most recent error that caused, respectively, the I/O thread or SQL thread to stop. During normal replication, these fields are empty. If an error occurs and causes a message to appear in either of these fields, the error values also appear in the error log.
- **Last\_IO\_Errno, Last\_SQL\_Errno:** The error number associated with the most recent error that caused, respectively, the I/O or SQL thread to stop. During normal replication, these fields contain the number 0.
- **Last\_IO\_Error\_Timestamp, Last\_SQL\_Error\_Timestamp:** The time stamp of the most recent error that caused, respectively, the I/O thread or SQL thread to stop, in the format YYMMDD HH:MM:SS. During normal replication, these fields are empty.

**Note:** If you are unable to log in to the slave using a valid account on the master, it might be because you created the user account with `CREATE USER` or `GRANT` on the master, and it replicated to the slave via the `mysql.user` table, but the replication process did not flush privileges. Use the `--flush-privileges` option with `mysqldump`, or execute `FLUSH PRIVILEGES` on the slave from another account.

The error log contains information about when replication begins and about any errors that occur during replication. For example, the following sequence shows a successful start and then a subsequent failure in replicating a user that already exists on the slave:

```
2012-12-16 11:13:21 8449 [Note] Slave I/O thread: connected to master  
'repl@127.0.0.1:3313',replication started in log 'FIRST' at position 4  
2012-12-16 11:13:21 8449 [Note] Slave SQL thread initialized, starting  
replication in log 'mysql-bin.000002' at position 108, relay log  
'./slave-relay-bin.000003' position: 408  
...  
2012-12-16 16:42:53 8449 [ERROR] Slave SQL: Error 'Operation CREATE  
USER failed for 'user'@'127.0.0.1' on query. Default database:  
'world_innodb'. Query: 'CREATE USER 'user'@'127.0.0.1' IDENTIFIED BY  
PASSWORD '*2447D497B9A6A15F2776055CB2D1E9F86758182F'', Error_code:  
1396  
2012-12-16 16:42:53 8449 [Warning] Slave: Operation CREATE USER failed  
for 'user'@'127.0.0.1' Error_code: 1396  
2012-12-16 16:42:53 8449 [ERROR] Error running query, slave SQL thread  
aborted. Fix the problem, and restart the slave SQL thread with "SLAVE  
START". We stopped at log 'mysql-bin.000002' position 460
```

The following sequence shows a failure of the I/O thread reading from the master's binary log:

```
2012-12-16 16:48:13 8823 [Note] Slave I/O thread: connected to master  
'repl@127.0.0.1:3313',replication started in log 'mysql-bin.000002' at position 1172  
2012-12-16 16:48:15 8823 [ERROR] Read invalid event from master: 'Found invalid event  
in binary log', master could be corrupt but a more likely cause of this is a bug  
2012-12-16 16:48:15 8823 [ERROR] Slave I/O: Relay log write failure: could not queue  
event from master, Error_code: 1595  
2012-12-16 16:48:15 8823 [Note] Slave I/O thread exiting, read up to log 'mysql-  
bin.000003', position 4
```

# Troubleshooting MySQL Replication

- Use **mysqlrplcheck** to ensure that servers meet the prerequisites for replication.
- Issue a **SHOW PROCESSLIST** command on the master and slave.
  - Review the state of the Binlog dump, I/O, and SQL threads.
- For a slave that suddenly stops working, check the most recently replicated statements.
  - The SQL thread stops if an operation fails due to a constraint problem or other error.
    - The error log contains events that cause the SQL thread to stop.
  - Review known replication limitations.
    - <http://dev.mysql.com/doc/mysql/en/replication-features.html>
  - Verify that the slave data has not been modified directly (outside of replication).

## SHOW PROCESSLIST on Slave

I/O thread states connecting to master:

- Verify privileges for the user being used for replication on the master.
- Verify that the hostname and port are correct for the master.
- Verify that networking has not been disabled on the master or the slave (with the `--skip-networking` option).
- Attempt to ping the master to verify that the slave can reach the master.

For more information about known replication limitations and other issues that can affect replication, see the *MySQL Reference Manual* at:

<http://dev.mysql.com/doc/mysql/en/replication-features.html>

## **Summary**

In this lesson, you should have learned how to:

- Describe MySQL replication
- Manage the MySQL binary log
- Explain MySQL replication threads and logs
- Set up a MySQL replication environment
- Explain the role of replication in high availability and scalability
- Design advanced replication topologies
- Perform a controlled switchover
- Configure replication with MySQL Utilities
- Monitor MySQL replication
- Troubleshoot MySQL replication

## **Chapter 12**

### **Introduction to Performance Tuning**

# **Objectives**

After completing this lesson, you should be able to:

- List factors that affect performance
- Describe general table optimizations
- Analyze queries by using EXPLAIN and PROCEDURE ANALYSE
- Monitor the most common status variables that affect performance
- Explain the most common server system variables that affect performance
- Describe the process of tuning system variables

# Factors That Affect Performance

- Environmental issues
  - CPU
  - Disk I/O
  - Network performance
  - Operating system contention
- MySQL configuration
  - Database design
    - Indexes, data types, normalization
  - Application performance
    - Specific requests, short transactions
  - Configuration variables
    - Buffers, caches, InnoDB settings

The performance of MySQL is affected by the performance characteristics of the host. Various factors can affect the performance of the host: CPU speed and number, disk throughput and access time, network throughput, and competing services on the operating system all have some bearing on the performance of a MySQL instance.

The database contents and its configuration also have an effect on MySQL's performance.

- Databases that have frequent small updates benefit by being well designed and normalized.
- Database throughput improves when you use the smallest appropriate data types to store data.
- Queries that request only a subset of table data benefit from well-designed indexes.
- Applications that request only specific rows and columns reduce the overhead that redundant requests create.
- Shorter transactions result in fewer locks and delays in other transactions.
- Well-tuned server variables optimize the allocation of MySQL's buffers, caches, and other resources for a specific workload and data set.

# Monitoring

- Benchmarking
  - **mysqlslap**
  - **sql-bench**
- Profiling
  - Logs
    - General query log
    - Slow query log
  - Statements
    - **EXPLAIN**
    - **PROCEDURE ANALYSE**
  - **SHOW STATUS**
    - Also visible with **mysqladmin extended-status**
  - **PERFORMANCE\_SCHEMA** database

To tune your server's performance, you must understand its performance characteristics. You can do this by benchmarking its overall performance, and by profiling events either individually with logs and EXPLAIN or as a group using PERFORMANCE\_SCHEMA.

MySQL installations provide the following benchmarking tools:

- **mysqlslap** is part of the standard MySQL distribution. It is a diagnostic program that emulates client load on a MySQL Server instance, and displays timing information of each stage.
- **sql-bench** is part of the MySQL source distribution, and is a series of Perl scripts used to perform multiple statements and gather status timing data.

The following is an example of using **mysqlslap** to set up a schema from a SQL script and run queries from another script:

```
mysqlslap --iterations=5000 --concurrency=50 --query=workload.sql  
--create-schema.sql --delimiter=";"
```

The slow query log records statements that exceed limits set by the `long_query_time` and `min_examined_row_limit` variables. Use `mysqldumpslow` to view the contents of the slow query log. The general query log records all client connections and requests received by MySQL. Use it to record all SQL statements received during a period of time (for example, to generate a workload for the use of `mysqlslap` or other benchmarking tools).

Third-party benchmarking suites are also available.

# Performance Schema

*Performance Schema* is a feature for monitoring MySQL Server execution at a low level.

- It is implemented using the PERFORMANCE\_SCHEMA storage engine and the performance\_schema database.
- Performance Schema monitors and allows you to inspect performance characteristics of instrumented code in MySQL Server.
  - Functions and other coded events are instrumented by their developers to collect timing information.
  - The exposed performance data is useful for:
    - Contributors to the MySQL codebase
    - Plug-in developers
    - Identifying low-level performance bottlenecks, such as log file I/O waits or buffer pool mutexes

Performance Schema is available in all binary versions of MySQL downloaded from Oracle.

Performance Schema is enabled by default, and controlled at server startup with the performance\_schema variable. Verify that Performance Schema is enabled with the following statement:

```
mysql> SHOW VARIABLES LIKE 'performance_schema' ;  
+-----+-----+  
| Variable_name | Value |  
+-----+-----+  
| performance_schema | ON |  
+-----+-----+
```

Performance Schema is designed to run with minimal overhead. The actual impact on server performance depends on how it is configured. The information Performance Schema exposes can be used to identify low-level bottlenecks. This is extremely useful for developers of the MySQL Server product family when debugging performance problems, and for system architects and performance consultants when tuning InnoDB data and log file storage hardware.

# Instruments, Instances, Events, and Consumers

The Performance Schema database contains configuration and event information:

- *Instruments* are points in the server code that raise events for monitoring, and are configured in the `setup_instruments` table.
- Each instrumented object is an *instance* of that instrument, recorded in a series of instance tables.
- MySQL identifies *events* that occur when a thread executes code in the instrumented instances, recording them in event and summary tables.
- Each *consumer* is the name of a table in Performance Schema used to record and query events and summaries of events, and is configured in the `SETUP_CONSUMERS` table.

Instruments in Performance Schema are points within the server source code from which MySQL raises events. Instruments have a hierarchical naming convention. For example, the following is a short list of some of the hundreds of instruments in Performance Schema:

```
stage/sql/statistics
statement/com/Binlog Dump
wait/io/file/innodb/innodb_data_file
wait/io/file/sql/binlog
wait/io/socket/sql/server_unix_socket
```

Each instrument consists of its type, the module it belongs to, and the variable or class of the particular instrument. View all available instruments by querying the `performance_schema.setup_instruments` table.

Performance Schema records each instrumented instance in an instance table. For example, the following query shows that the instrument `wait/io/file/sql/FRM` records events on the file instance `/var/lib/mysql/mem/tags.frm`.

```
mysql> SELECT file_name, event_name FROM file_instances LIMIT 1\G
***** 1. row *****
FILE_NAME: /var/lib/mysql/mem/tags.frm
EVENT_NAME: wait/io/file/sql/FRM
```

The following output shows the contents of the `setup_consumers` table:

```
mysql> SELECT * FROM setup_consumers;
+-----+-----+
| NAME           | ENABLED |
+-----+-----+
| events_stages_current | NO      |
| events_stages_history | YES     |
| events_stages_history_long | NO      |
| events_statements_current | YES     |
| events_statements_history | NO      |
| events_statements_history_long | NO      |
| events_waits_current | YES     |
| events_waits_history | YES     |
| events_waits_history_long | NO      |
| global_instrumentation | YES     |
| thread_instrumentation | YES     |
| statements_digest | YES     |
+-----+-----+
12 rows in set (0.00 sec)
```

Each consumer `NAME` is the name of a table in Performance Schema used to query events and summaries. A disabled consumer does not record information, which saves system resources.

As MySQL identifies events that occur in the instrumented instances, it records them in event tables.

- The primary event table is `events_waits_current`, which stores the most recent event for each thread.
- `events_waits_history` stores the 10 most recent events for each thread.
- `events_waits_history_long` stores the 10,000 most recent events in total.

The `events_waits_*` tables all have the same schema. For information about how that schema is structured, see <http://dev.mysql.com/doc/mysql/en/events-waits-current-table.html>.

When using Performance Schema to identify a bottleneck or other problem, do the following:

1. Ensure that you have enabled Performance Schema with a range of instruments and consumers applicable to the type of problem that you are experiencing. For example, use the `wait/io/file/*` instruments if you are sure the problem is I/O bound, or a wider range if you are unsure of the root cause.
2. Run the test case that produces the problem.
3. Query consumers such as the `events_waits_*` tables, in particular `events_waits_history_long` with suitable `WHERE` clause filters to further narrow the cause of the problem.
4. Disable instruments that measure problems you have ruled out.
5. Retry the test case.

# General Database Optimizations

- Normalize the data to:
  - Eliminate redundant data
  - Improve performance of transactional workloads
  - Provide flexible access to data
  - Minimize data inconsistencies
- Choose the correct data types and size to:
  - Avoid NULLS
  - Improve performance
  - Protect data
  - Use data compression where appropriate
- Create the optimal indexes to:
  - Improve query throughput
  - Reduce I/O overhead

## Normalization

Normalization is the act of removing redundancies and improper dependencies in the database to avoid storing the same data in multiple places and risking anomalies.

Normalization generally results in more tables with fewer columns, lower overall storage requirements, lower I/O requirements, and faster individual inserts, updates, and deletes. This improves the performance of transactional workloads that perform frequent small updates, but can complicate queries that retrieve large amounts of data.

## Data Types and Size

Choosing the correct data type is an important but often overlooked part of table design, yet the size of a data type has a large potential effect on table operations. For example, choosing to store a `SMALLINT` number as an `INT` doubles the space required by that column. In a table of a million rows, that decision results in an extra 2 MB in storage wasted, along with slower disk operations, and more memory used by buffers and caches.

Use `INSERT ... COMPRESS(field_name) ...` and `SELECT ... UNCOMPRESS(column_name)` ... to compress and uncompress string data while storing and retrieving it. Although you can use `CHAR` or `VARCHAR` fields for this purpose, avoid problems with character set conversion by using `VARBINARY` or `BLOB` columns to store compressed data.

Also consider how columns are compared in queries. If the MySQL server must convert data types to compare them, there is a performance overhead. For example, when you compare or join character fields that use different collations, MySQL must coerce one of them to match the other, which slows down the query. Similarly, avoid queries that compare numeric values to other numeric values, and not string types.

## **Efficient Indexes**

When you query a table for specific rows by specifying a field in the `WHERE` clause, and that table does not have an index on that field, MySQL reads every row in that table to find each matching row. This results in a lot of unnecessary disk access and can greatly slow down the performance of large tables.

Indexes are ordered sets of data that make it easier for MySQL to find the correct location of a queried row. InnoDB orders the table according to the primary key by default; this ordered table is called a clustered index. Every additional or secondary index on an InnoDB table takes up extra space on the file system because the index contains an extra copy of the fields it indexes, along with a copy of the primary key.

Every time you modify data with an `INSERT`, `UPDATE`, `REPLACE`, or `DELETE` operation, MySQL must also update all indexes that contain the modified fields. As a result, adding many indexes to a table reduces the performance of data modifications that affect that table.

However, properly designed indexes produce large gains in the performance of queries that rely on the indexed fields. A query that cannot use an index to find a particular row must perform a full table scan; that is, it must read the entire table to find that row. A query that uses an index can read the row directly without reading other rows, which greatly speeds up the performance of that type of query.

To easily identify queries that expect to retrieve all rows and could therefore potentially benefit from adding indexes to the tables involved, enable the slow query log and the `--log-queries-not-using-indexes` server option. Note that this option also logs any query that uses an index but performs a full index scan.

## PROCEDURE ANALYSE

- PROCEDURE ANALYSE analyzes the columns in a given query and provides tuning feedback on each field:

```
mysql> SELECT CountryCode, District, Population  
-> FROM City PROCEDURE ANALYSE(250,1024)\G
```

- Provide arguments to tune how it suggests ENUMS:
  - The maximum number of elements and memory to use when processing the column
  - Example: **...PROCEDURE ANALYSE(100,256);**
- Use it to minimize field size.
  - MySQL often allocates memory for the maximum field size.
  - Implicit MEMORY temp tables, sort buffer, and so on
- Use it to determine whether a field should allow NULL.

The default settings often suggest ENUM types to optimize a table's design. If you are sure you do not want to use an ENUM value when analyzing a column for which PROCEDURE ANALYSE() suggests it, use non-default arguments.

- The first argument is the number of distinct elements to consider when analyzing to see whether ENUM values are appropriate. This argument has a default value of 256.
- The second argument is the maximum amount of memory used to collect distinct values for analysis. This argument has a default of 8192, representing 8 KB. If you set a value of 0 for this argument, PROCEDURE ANALYSE() cannot examine distinct values to suggest an ENUM type.

If PROCEDURE ANALYSE() cannot store an acceptable range of candidate ENUM values within the limits set by its arguments, it does not suggest an ENUM type for that column.

The example in the slide suggests a CHAR(3) type for the City.CountryCode column. On the other hand, if you use the default arguments, PROCEDURE ANALYSE() suggests ENUM('ABW', 'AFG', ..., 'ZMB', 'ZWE'), an ENUM type with over 200 elements containing a distinct value for each corresponding CountryCode value.

## **EXPLAIN**

- **EXPLAIN command**
  - Describes how MySQL intends to execute your particular SQL statement
  - Does not return any data from the data sets
  - Provides information about how MySQL plans to execute the statement
- Use EXPLAIN to examine SELECT, INSERT, REPLACE, UPDATE, and DELETE statements.
- Precede your statement with **EXPLAIN**:
  - **EXPLAIN SELECT ...**
  - **EXPLAIN UPDATE...**

EXPLAIN produces a row of output for each table used in the statement. The output contains the following columns:

- **table**: The table for the output row
- **select\_type**: The type of select used in the query. SIMPLE means the query does not use UNION or subqueries.
- **key**: The index chosen by the optimizer
- **ref**: The columns compared to the index
- **rows**: Estimated number of rows that the optimizer examines
- **Extra**: Additional information provided per-query by the optimizer

For a complete discussion of the output columns, see:

<http://dev.mysql.com/doc/mysql/en/explain-output.html>

Use EXPLAIN EXTENDED . . . to view additional information provided by the optimizer.

For example, the following query joins fields from two tables and performs an aggregation:

```
mysql> SELECT COUNT(*) as 'Cities', SUM(Country.Population) AS Population,
    > Continent FROM Country JOIN City ON CountryCode = Code
    > GROUP BY Continent ORDER BY Population DESC;

+-----+-----+-----+
| Cities | Population | Continent |
+-----+-----+-----+
| 1765   | 900934498400 | Asia      |
| 580    | 95052481000  | North America |
| 842    | 55127805400  | Europe     |
| 470    | 48533025000  | South America |
| 366    | 16179610000  | Africa     |
| 55     | 307500750   | Oceania    |
+-----+-----+-----+
6 rows in set (0.01 sec)
```

The following output shows the result of using EXPLAIN with the preceding query:

```
mysql> EXPLAIN SELECT COUNT(*) as 'Cities', SUM(Country.Population) AS Population,
    > Continent FROM Country JOIN City ON CountryCode = Code
    > GROUP BY Continent ORDER BY Population DESC\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: Country
        type: ALL
possible_keys: PRIMARY
        key: NULL
    key_len: NULL
        ref: NULL
       rows: 239
    Extra: Using temporary; Using filesort
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: City
        type: ref
possible_keys: CountryCode
        key: CountryCode
    key_len: 3
        ref: world_innodb.Country.Code
       rows: 9
    Extra: Using index
2 rows in set (0.00 sec)
```

The EXPLAIN output lists the joined tables in the order they are read. EXPLAIN approximates the total number of rows that must be read from each table. The total approximate number of rows that this SELECT query must examine is the product of the rows examined in each joined table:

239 x 9 = 2151 rows.

## EXPLAIN Formats

EXPLAIN output is available in other formats:

- Visual EXPLAIN:
  - Outputs in a graphical format
  - Is available in MySQL Workbench
- **EXPLAIN FORMAT=JSON:**
  - Outputs in JSON format
  - Is useful when passing EXPLAIN output to programs for further processing/analysis

JSON (JavaScript Object Notation) is a simple data interchange format. The following output shows the result of using `FORMAT=JSON` in an `EXPLAIN` statement:

```
mysql> EXPLAIN FORMAT=JSON SELECT COUNT(*) as 'Cities',
    SUM(Country.Population) AS Population, Continent FROM Country JOIN City
    ON CountryCode = Code GROUP BY Continent ORDER BY Population DESC\G
*****
1. row *****

EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "ordering_operation": {
      "using_filesort": true,
      "grouping_operation": {
        "using_temporary_table": true,
        "using_filesort": false,
      }
    }
  }
}
...
1 row in set, 1 warning (0.00 sec)
```

# Examining Server Status

MySQL provides several ways to view server status variables:

- At the mysql prompt:
  - **STATUS;**
  - **SHOW STATUS;**
- At the terminal:
  - **mysqladmin --login-path=login-path status**
  - **mysqladmin -u user -p extended-status**

MySQL provides a short status message with the mysql command STATUS and the mysqladmin command status. The longer form status output, shown with the mysql command SHOW STATUS and the mysqladmin command extended-status contains values for many system status variables, the most important of which are discussed in the following slides.

Use options with mysqladmin to provide additional functionality. For example, the **--sleep** (or **-i**) option specifies the number of seconds to wait between iterations, and automatically re-executes the command after that time. The **--relative** (or **-r**) option displays the difference in each variable since the last iteration, rather than its value.

Use command-line tools such as grep to extend how you use mysqladmin. For example, use the following command to show only variables that contain the string `cache_hits`:

```
shell> mysqladmin --login-path=admin extended-status | grep cache_hits
| Qcache_hits                                | 0      |
| Ssl_callback_cache_hits                     | 0      |
| Ssl_session_cache_hits                      | 0      |
| Table_open_cache_hits                       | 280    |
```

# Top Status Variables

- **Created\_tmp\_disk\_tables:** Displays the number of internal on-disk temporary tables
- **Handler\_read\_first:** Displays the number of times the first entry in an index was read
- **Innodb\_buffer\_pool\_wait\_free:** Displays the number of times the server waits for a clean page
- **Max\_used\_connections:** Displays the maximum number of simultaneous connections since the server started
- **Open\_tables:** Displays the number of open tables at a given time

**Created\_tmp\_disk\_tables:** For the temporary tables created by the server while executing statements. If this number is high, the server has created many temporary tables on disk rather than in memory, resulting in slower query executions.

**Handler\_read\_first:** If this number is high, the server has performed many full index scans to complete query requests.

**Innodb\_buffer\_pool\_wait\_free:** Wait for a page to be flushed in the InnoDB buffer pool before the query request can be completed. If this number is high, the InnoDB buffer pool size is not correctly set and query performance suffers.

**Max\_used\_connections:** This variable provides valuable information for determining the number of concurrent connections that your server must support.

**Open\_tables:** When compared with the `table_cache` server system variable, this provides valuable information about the amount of memory that you should set aside for the table cache. If the value of the `Open_tables` status variable is usually low, reduce the size of the `table_cache` server system variable. If its value is high (approaching the value of the `table_cache` server system variable), increase the amount of memory assigned to the table cache to improve query response times.

# Top Status Variables

- **Select\_full\_join:** Displays the number of joins that perform table scans instead of using indexes
- **Slow\_queries:** Displays the number of queries that have taken longer than the number of seconds specified by the **long\_query\_time** system variable
- **Sort\_merge\_passes:** Displays the number of merge passes that the sorting algorithm has performed
- **Threads\_connected:** Displays the number of currently open connections
- **Uptime:** Displays the number of seconds that the server has been up

**Select\_full\_join:** If this value is not 0, you should carefully check the indexes of your tables.

**Slow\_queries:** This status variable depends on knowing the setting of the **long\_query\_time** variable, which defaults to 10 seconds. If the **Slow\_queries** status variable is not 0, check the value of the **long\_query\_time** and the slow query log and improve the queries that have been captured.

**Sort\_merge\_passes:** Sorting operations require a buffer in memory. This status variable counts the passes through sort buffers that sort operations require. If this value is high, it can indicate that the sort buffer size is not sufficient to perform one-pass sorting of queries; consider increasing the value of the **sort\_buffer\_size** system variable.

**Threads\_connected:** Capturing this value on a regular basis provides you with valuable information about when your server is most active. Use this variable to determine the best time to perform maintenance on the server, or as a justification for allocating more resources to the server.

**Uptime:** This value can provide valuable information about the health of the server, such as how often the server needs to be restarted.

# Tuning System Variables

- Tune queries, schema, and indexes first.
  - More benefit per effort than tuning variables
- Tune for server size.
  - Memory and I/O
- Tune for application configuration.
  - Storage engine settings
    - Give 70%–85% of physical RAM to InnoDB buffer pool
    - Minimize MyISAM caches and buffers
- Tune for workload type.
  - Number of connections
  - Transactional server
  - Reporting server

There is a common misconception that server variable configuration is the most important part of server tuning. In fact, in terms of effort spent, more benefits come from optimizing the schema, common queries, and indexes of a typical database than from tuning variables.

## Default Settings

The default settings have been chosen by Oracle's MySQL engineers to suit a majority of production systems, which tend to have frequent small transactions, many updates and few large, slow queries such as those used to generate reports. However, because MySQL is used on systems from small devices (such as point-of-sale systems and routers) to large web servers with huge amounts of memory and fast disk arrays, you may find that your particular environment and workload benefits from changing some server settings from the default.

## InnoDB Settings

For example, on a server that is dedicated to MySQL using only InnoDB user tables, increase the value of `innodb_buffer_pool_size` to a large percentage of total server memory (70%–85%), bearing in mind the needs of the operating system such as cron jobs, backups, virus scans, and administrative connections and tasks. If you have several gigabytes of RAM, you might also benefit from having several `innodb_buffer_pool_instances`, a setting that enables multiple buffer pools to avoid contention.

## **Reducing MyISAM Settings**

On a system that does not use MyISAM for user tables, reduce the value of MyISAM-only options such as `key_buffer_size` to a small value such as 16 MB, bearing in mind that some internal MySQL operations use MyISAM.

## **Reporting Systems**

On servers used for running few large slow queries such as those used in business intelligence reports, increase the amount of memory dedicated to buffers with settings such as `join_buffer_size` and `sort_buffer_size`. Although the default server settings are better suited to transactional systems, the default `my.cnf` file contains alternate values for these variables, suited to reporting servers.

## **Transactional Systems**

On servers used to support many fast concurrent transactions that disconnect and reconnect repeatedly, set the value of `thread_cache_size` to a large enough number so that most new connections use cached threads; this avoids the server overhead of creating and tearing down threads for each connection.

On servers that support many write operations, increase log settings such as `innodb_log_file_size` and `innodb_log_buffer_size`, because the performance of data modifications relies heavily on the performance of the InnoDB log. Consider changing the value of `innodb_flush_log_at_trx_commit` to increase the performance of each commit at the risk of some data loss if the server fails.

If your application executes the same query (or many identical queries) repeatedly, consider enabling the query cache and sizing it according to the results of the common queries by setting appropriate values for `query_cache_type` and `query_cache_size`.

## **Balancing Memory Use**

When you set larger values for per-query or per-connection caches and buffers, you reduce the available size for the buffer pool. Tuning a server's configuration variables is a balancing process where you start with the defaults, give as much memory as you can to the buffer pool, and then tune the variables that are most closely associated with your goals for tuning, problems that you identify from examining the server status, and bottlenecks that you identify by querying Performance Schema.

# Top Server System Variables

- **innodb\_buffer\_pool\_size**: Defines the size (in bytes) of the memory buffer that InnoDB uses to cache data and indexes of its tables
- **innodb\_flush\_log\_at\_trx\_commit**: Defines how often InnoDB writes the log buffer to the log file, and how often it performs the flush-to-disk operation on the log file
- **innodb\_log\_buffer\_size**: Defines the size (in bytes) of the buffer that InnoDB uses to write to the log files on disk
- **innodb\_log\_file\_size**: Defines the size (in bytes) of each log file in a log group

**innodb\_buffer\_pool\_size**: For best performance, set this value as large as possible, bearing in mind that too high a value causes the operating system to swap pages, which slows down performance considerably. If you use only InnoDB user tables on a dedicated database server, consider setting this variable to a value from 70% to 85% of physical RAM.

**innodb\_flush\_log\_at\_trx\_commit**: There are three possible settings for this variable:

- 0: Write the log buffer to disk once per second.
- 1: Flush the log to disk at each commit, or once per second if no commit occurs.
- 2: Flush the log to the operating system cache, and flush to disk every `innodb_flush_log_at_timeout` seconds (with a default of one second)

**innodb\_log\_buffer\_size**: This variable has a default value of 8 MB. Transactions that exceed this size cause InnoDB to flush the log to disk before the transaction commits, which can degrade performance. For applications that use a large number of BLOBs or that have a large spike in update activity, improve transaction performance by increasing this value.

**innodb\_log\_file\_size**: For write-intensive workloads on large data sets, set this variable so that the total maximum size of all log files (as set by `innodb_log_files_in_group`) is less than or equal to the size of the buffer pool. Larger log files slow down crash recovery, but improve overall performance by reducing checkpoint flush activity.

# Top Server System Variables

- **join\_buffer\_size**: Defines the minimum size of the buffer used for joins that use table scans
- **query\_cache\_size**: Defines the amount of memory allocated for caching query results
- **sort\_buffer\_size**: Defines the maximum amount of memory allocated to sessions that need to do a sort
- **table\_open\_cache**: Defines the number of open tables for all threads
- **thread\_cache\_size**: Defines the number of threads that the server should cache for reuse

**join\_buffer\_size**: Increase this value from its default (256 KB) for queries that have joins that cannot use indexes. Change the per-session value when running such queries to avoid setting the global setting and wasting memory for queries that do not need such a large value.

**query\_cache\_size**: Improve the performance of applications that issue repeated queries on data that rarely changes by using the query cache. As a baseline, set this variable to a value from 32 MB to 512 MB based on the number of repeated queries and the size of data they return. Monitor the cache hit ratio to determine the effectiveness of this variable, and tune its value based on your observations.

**sort\_buffer\_size**: Increase this value to improve the performance of ORDER BY and GROUP BY operations if the value of the `Sort_merge_passes` status variable is high.

**table\_open\_cache**: Set this value so that it is larger than  $N * \text{max\_connections}$ , where  $N$  is the largest number of tables used in any query in your application. Too high a value results in the error “Too many open files.” High values of the `Open_tables` status variable indicate that MySQL frequently opens and closes tables, and that you should increase `table_open_cache`.

**thread\_cache\_size**: By default, this variable is sized automatically. Evaluate the `Threads_created` status variable to determine whether you need to change the value of `thread_cache_size`.

# Preparing for Tuning

Prepare the tuning environment:

- Duplicate the production system as much as possible.
- Decide on a tuning goal.
  - More transactions per second
  - Faster generation of complex reports
  - Improved performance with peak numbers of concurrent connections
- Choose appropriate variables to tune.
  - Buffers, caches, log settings
- Collect representative statements.
  - Application code
  - General query log

Tuning a database server can be compared to tuning a musical instrument:

- Choose the value that you want to change, and identify a goal.
- Test the behavior of the instrument while adjusting the value up and down.
- Identify the optimal setting.

To reduce the impact of changed factors unrelated to the variables being tuned, perform tuning on the production server during down time, or preferably on a duplicate system.

Before tuning, decide on a goal. The variables you choose to tune depend on the goal you set.

The optimal settings for a reporting server with few connections are very different to those of a transactional application server with many connections and hundreds of small transactions per second. A server that has a high ratio of memory to database size has very different performance characteristics to one with less memory but a larger database. Write-heavy workloads require different settings to read-only systems.

To most accurately model the workload for which you are tuning, gather a representative set of statements. Choose a sequence of statements from the application that has the correct proportion of queries and modifications. Use the general query log to gather actual statements from the production server during the daily or weekly period for which you wish to optimize.

## Tuning in Practice

Benchmark to find the optimal value for each variable:

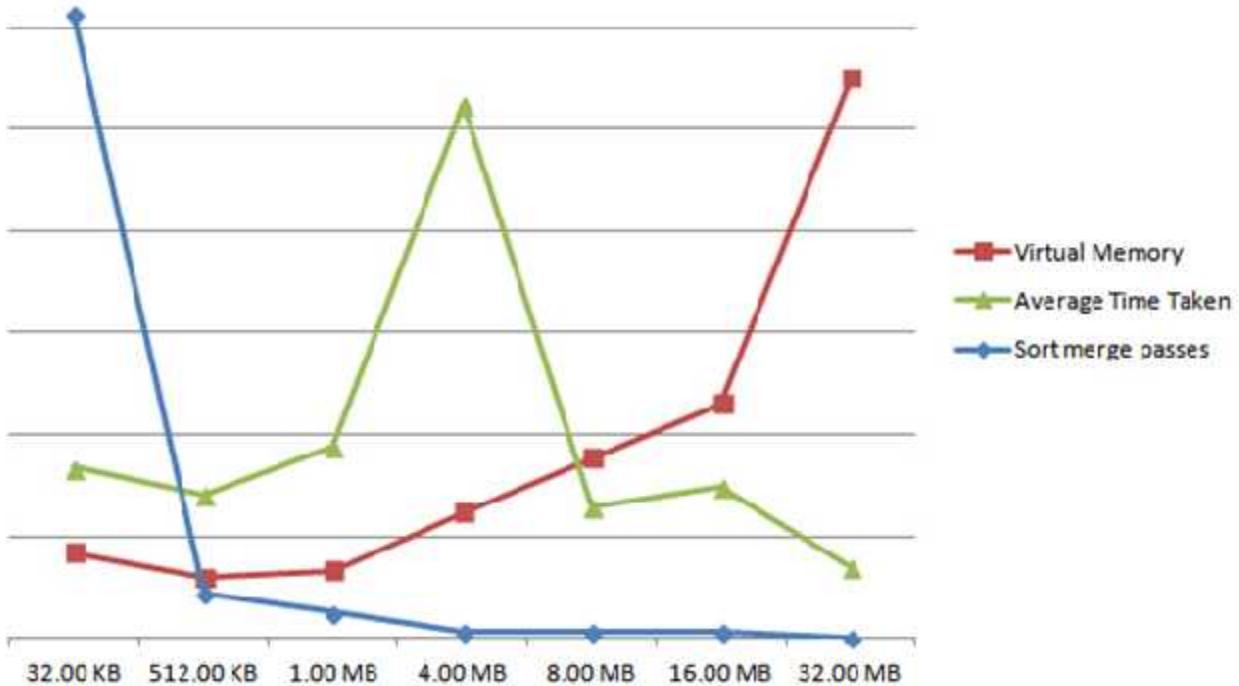
- Set the variable to a setting lower than its default value.
- Benchmark, measuring relevant metrics:
  - Virtual memory use
  - Average time taken
  - Relevant status variables
- Increase the variable value, and repeat the benchmarks.
  - Flush the status variables, if required.
- Graph the results.
  - Look for fall-offs in benefits and peaks in performance.
  - Decide the final variable value based on the best-case trade-off between resources used and performance.

To view the values of your chosen metrics, use:

- `mysqlslap` or `mysql` to run the workload and get the average execution time
- `sql-bench` to run more general benchmarks
- `mysqladmin extended-status` to get the values of status variables before and after the workload
- Operating system tools such as `top` or the `/proc` file system to access process metrics

If you want to run fine-tuned benchmarks with many different values for a particular variable, or if you want to run identical benchmarks repeatedly over a long period, consider using a scripting language to automate the steps used in the benchmark.

## Tuning Example: Sort Buffer Size



The example in the slide shows the result of a series of tests against a database with a sort-heavy workload, where the `sort_buffer_size` variable changes between test runs.

The diagram shows:

- The value of the `Sort_merge_passes` status variable (viewed with `mysqladmin extended_status -r`) falling off dramatically with an increase in `sort_buffer_size` from 32 KB to 512 KB, tailing off gradually after that
- The average time taken (viewed with `mysqlslap`) for the test workload falling at a `sort_buffer_size` of 512 KB, peaking significantly at 4 MB, then falling off at 8 MB with an eventual best performance at 32 MB
- The total virtual memory of the `mysqld` process (viewed with `top`) has a minimum at a `sort_buffer_size` of 512 KB, rising steadily after that until 16 MB, with a sharp rise at 32 MB

While the average time of queries is at its lowest with a `sort_buffer_size` of 32 MB, that setting uses a lot of memory that the buffer pool could put to better use. In this example, a setting of 512 KB provides the best trade-off between performance and memory use for the specific combination of workload, server, and database used in the test.

## **Summary**

In this lesson, you should have learned how to:

- List factors that affect performance
- Describe general table optimizations
- Analyze queries by using EXPLAIN and PROCEDURE ANALYSE
- Monitor the most common status variables that affect performance
- Explain the most common server system variables that affect performance
- Describe the process of tuning system variables

## **Chapter 13**

### **Introduction to Mysql Cluster**

# What Is a Cluster?

A *cluster* is:

- A group of computers working in a coordinated way to perform a function, in hardware or software
- Designed for high performance or high availability, or both

MySQL Cluster:

- Distributes data across multiple cluster hosts
- Enables high performance by allowing parallel reads and writes on multiple hosts
- Enables high availability by storing data replicas on multiple hosts

In general, a cluster consists of a number of computers that are connected via a fast local network and coordinated by a centralized management process. The exact clustering mechanism depends on the implementation. Some clusters use shared resources such as shared memory or a shared clustered file system, while others (such as MySQL Cluster) use a message-passing protocol to keep nodes synchronized.

The MySQL Cluster architecture is described as “shared-nothing” because each node operates on hardware and storage that are independent of other nodes. (See the next slide for details about the “shared-nothing” architecture.)

Although MySQL Cluster stores data replicas, it uses a mechanism unlike MySQL’s replication feature, which is asynchronous or semi-synchronous. MySQL Cluster maintains connections between data nodes by using high-speed interconnects over TCP/IP standard or direct connections or Scalable Coherent Interface (SCI) sockets.

# Shared-Nothing Architecture

MySQL Cluster uses a *shared-nothing* architecture:

- Each node is self-sufficient.
- There is no single point of failure.
- Other architectures use shared resources:
  - Shared storage (such as disks)
  - Shared memory
  - Shared peripherals
- Shared-resource architectures are subject to a shared point of failure.

MySQL Cluster is a shared-nothing cluster, with no single point of failure. This means that each cluster node in a cluster has its own independent hardware and storage, so that any hardware failure causes only one node to fail and leaves the rest of the cluster unaffected.

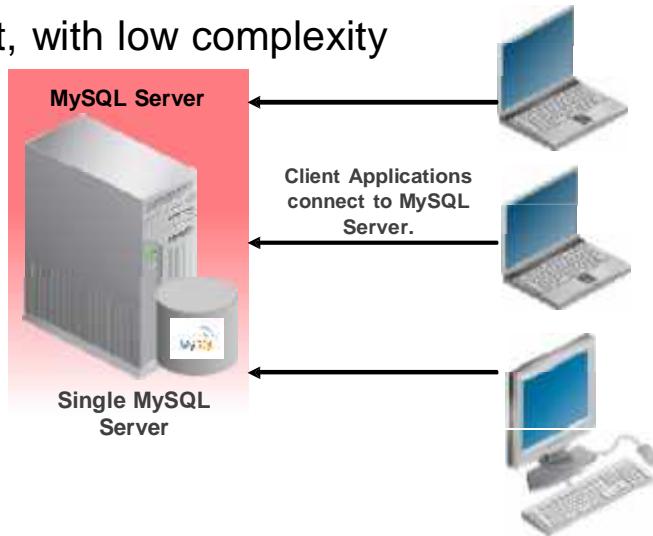
In a shared-disk architecture, nodes access a central storage location. In comparison to a shared-nothing architecture, a shared-disk system requires less network communication if it uses special hardware (such as a Fibre Channel SAN) for the shared storage. Some other shared-disk solutions use iSCSI, which requires special configuration and also requires high levels of network communication. In a virtualized environment, nodes can access a shared physical disk.

**Note:** If you run several MySQL Cluster data nodes on the same physical machine, the architecture is no longer shared-nothing.

## MySQL Server on a Single Server (Not Clustered)

Clients connect to a single server:

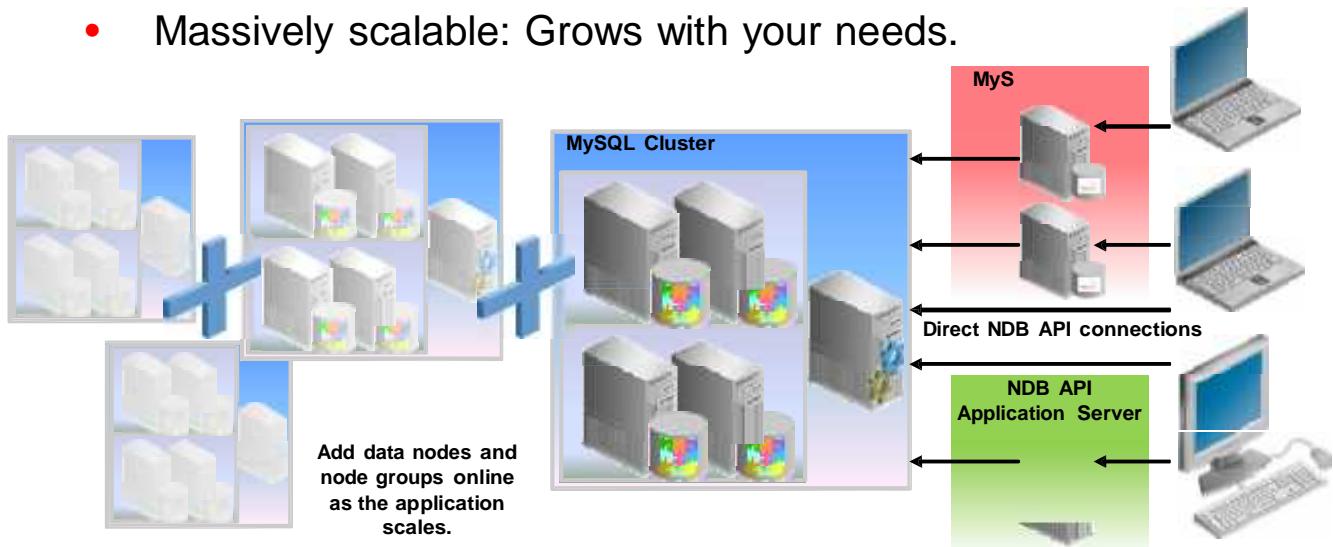
- There is no load-balancing, scalability, or high availability.
- All requests are handled by a single server.
- Easy to implement, with low complexity



Without MySQL Cluster, a MySQL Server instance stores data locally and serves that data to clients. The storage engine is typically InnoDB, which provides high performance and resilience but does not scale out to multiple machines and provides no redundancy.

# MySQL Cluster

- Each node accepts reads and writes.
- Data is partitioned across nodes for high performance.
- Data nodes contain partition replicas for high availability.
- Massively scalable: Grows with your needs.



In a MySQL Cluster environment, all data nodes accept writes, and all writes propagate immediately to all other data nodes. This means that both reads and writes scale with the number of data nodes.

MySQL Cluster automatically shards table data between and within node groups to improve performance. Nodes within a node group replicate partitions for high availability.

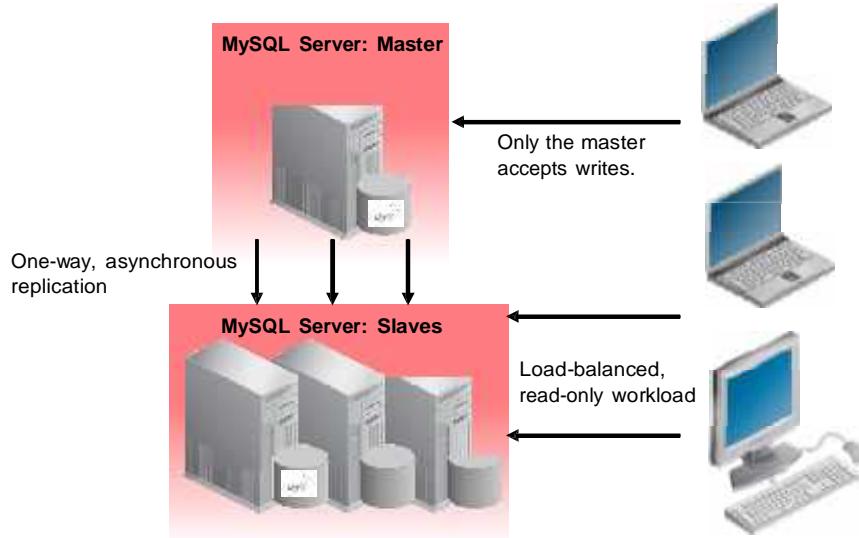
Applications that rely on SQL queries can continue to use MySQL Server as an intermediary layer; MySQL Cluster provides the NDB storage engine, which is also known as “NDBCLUSTER.” (NDB is an abbreviation for “network database.”) The NDB storage engine communicates with the cluster so that all servers that use any clustered table will share that table’s data.

Other API nodes can communicate directly with the cluster by using one of the many interfaces, from high-level JPA-based persistence layers and the Memcached NoSQL API to direct connections from high-performing applications using the NDB API.

Add additional data nodes, management nodes, and API nodes without any down time. Both read and write workloads scale with the addition of commodity hardware.

## MySQL Server in a Replicated Environment (Not Clustered)

- Failover from slave to master enables high availability.
- Clients can use load-balanced slaves for reading.
- Clients can write to only one server.



MySQL replication is one-way and asynchronous, and it uses binary log shipping. In a typical setup, clients write to the master and can read from either the master or any slave. You can configure client applications to operate in a load-balanced way, which means that read-only workloads are very scalable.

In the event of a master failing, you can promote a replication slave to become a master. This feature enables a basic form of high availability. With additional tools such as the `mysqlfailover` program from the MySQL Utilities suite, you can automate this type of failover.

# Anatomy of a MySQL Cluster

MySQL Cluster runs on a set of processes distributed across a network. Each process is called a *node*.

- Nodes run on cluster hosts.
  - Each host is a physical or virtualized server.
  - Multiple nodes can run on each host, although this is not ideal because it reduces redundancy.
- Management nodes are the central coordinators.
  - They manage the startup and configuration of cluster nodes.
- API nodes forward client requests to the data nodes.
  - MySQL Server instances can act as API nodes in a cluster.
- Data nodes host the cluster data and perform the main data storage and retrieval functions of the cluster.
  - Data nodes are grouped into parallel sets of sharded replicas.

A typical cluster consists of:

- One or more management nodes to handle configuration and node registration
- Data nodes organized into node groups of sharded replicas to handle the data
- API nodes to handle client requests

At a minimum, a cluster must have one node of each type. However, a minimal cluster does not have any of the key benefits of MySQL Cluster such as redundancy, failure protection, load balancing, or the efficiencies of data sharding.

Each node has its own configuration that includes the location of the management node. Shared configuration is handled by the management node.

# Management Nodes

- Read cluster configuration
  - Handle other nodes that are joining the cluster
  - Distribute configuration information to other nodes
  - Persist configuration of each node across node restarts
- Manage the cluster log
- Perform arbitration in the event of a network split
- Are controlled via a management client, which can:
  - Run backups
  - Start and stop other nodes
  - Check the cluster status
- Are redundant
  - You can run a cluster with one management node, but this reduces cluster resilience.



When a management server starts, it connects to another management server to read the cluster configuration. If there is no other management server, it reads its own configuration file.

When each node starts, it connects to the management server to retrieve configuration data. Cluster configuration is therefore kept centrally and is maintained by management nodes.

The cluster continues to run even if all management nodes fail, but no additional nodes can join the cluster until a management node reconnects.

## API Nodes

- Are the clients of a MySQL Cluster
- Applications connect in several ways:
  - Client applications or application servers
    - Connect directly using an API
    - Bypass the SQL layer for best performance
  - Applications using MySQL Server
    - Connect via NDB storage engine
- NoSQL connectors:
  - Direct connection from applications by using NDB API
  - ClusterJ and ClusterJPA for direct connection from Java
  - Node.js, Memcached, and Apache mod\_ndb for platform-targeted APIs



Standard MySQL applications that are written in Java, PHP, Python, Perl, and so on can use MySQL Cluster through a SQL node with no special additional cluster-specific programming knowledge. MySQL Server instances connect to the cluster by using the NDB storage engine, and applications can use such data transparently with no further configuration.

You can also create NoSQL applications by using one of several supported APIs.

ClusterJ is a high-level object-relational modeling (ORM) layer similar to Hibernate or the Java Persistence Architecture (JPA). ClusterJPA is an OpenJPA implementation that uses the NDB API or JDBC, depending on which is more effective for each specific purpose.

The JavaScript connector for Node.js enables connection between JavaScript applications and MySQL Cluster data without requiring a SQL layer. The Memcached NoSQL API is also mature and widely used. The Apache mod\_ndb module provides a web services API through a native HTTP/REST interface.

# Data Nodes

- Host the cluster's data
- Contain auto-sharded data
  - Are organized in replicas and fragments
- Can be organized in node groups
  - This provides redundancy (high availability).
  - Node groups contain partition replicas.
  - Each node group hosts a sharded partition.
- Store data in memory but can also store data on disk for large data volumes and other requirements
- Are usually hosted on separate physical instances
  - They can co-exist on a single host, although this reduces redundancy.



Data nodes are the software server components that host the cluster's data and manage its sharding and replication with other data nodes. Data nodes can be single-threaded or multi-threaded, with the latter variant recommended for data nodes on physical servers with multiple CPU cores.

By selecting a number of replicas of two or more, you configure data nodes in parallel node groups to enable high availability and best performance. Each node group contains one or more shards (partitions) of the data, and nodes within a node group contain replicas of those partitions. Failure of the last data node in a node group shuts down the cluster to maintain data consistency.

Data nodes usually store data in memory for best performance. Checkpoints record the data's state asynchronously so that it can be returned to memory after a restart. You can also configure disk-based storage to store large volumes of non-indexed data.

# Sharding

- Is the division of the content of a table into multiple shards or partitions
- Is a form of horizontal partitioning
- Automatically distributes rows across data nodes in the cluster in an efficient indexed way
- Results in smaller indexes for faster lookups
- Gives better memory usage and an overall improvement in performance when compared with non-sharded data
- Enables the cluster to scale database tables across more than one physical machine
- Supports more transactions per second than non-sharded systems because of parallel reads and writes



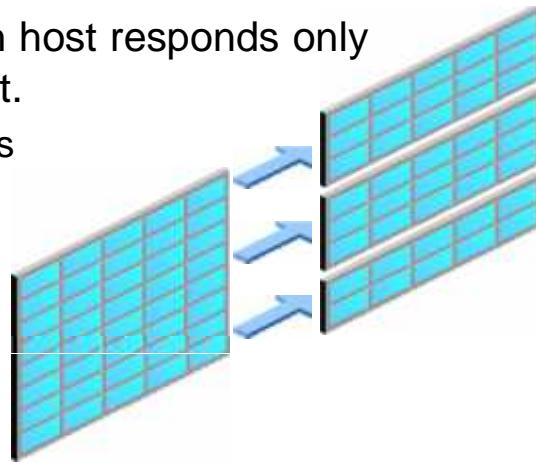
Sharding enables you to scale a database beyond the capacity of one machine because table contents are partitioned and partitions are distributed between data nodes.

Sharding also results in more transactions per second (TPS) because each node is responsible for only part of the data, so both read and write requests can be distributed between nodes and executed in parallel.

**Note:** Sharding is sometimes referred to as *auto-partitioning* or *auto-sharding*.

## Horizontal Partitioning

- MySQL Cluster partitions data horizontally across data nodes.
  - Partitioned by row rather than by column
- Each table is physically distributed across multiple nodes.
  - Each node is usually on a separate physical host.
- Partitioning ensures that each host responds only to queries of data on that host.
  - Only touches affected shards
  - Supports more TPS
  - Uses parallel execution



In MySQL Cluster, tables are automatically sharded using a technique known as *horizontal partitioning*, which distributes rows of data from a single table across multiple dispersed storage media. In MySQL Cluster, this separation takes place across multiple data nodes, each hosted on a separate physical device. The separate entities that are created in this way – and the individual columns that hold the data rows – all have the same attributes to ensure consistency across partitions.

A cluster that shards data across partitions supports more transactions per second because each transaction needs to touch only the partition (or partitions) that it affects and does not require resources from any other partitions. However, partitioning does not help with index caching because there are no on-disk indexes. In addition, partitioning provides parallel query execution, which happens within the data nodes rather than the SQL nodes.

## When to Choose MySQL Cluster

MySQL Cluster is ideal for some purposes but not for all.

- Use InnoDB on MySQL Server instances for:
  - “Hot” data sets that cannot fit entirely in RAM
  - Flexible transaction support
- Use MySQL Cluster through the NDB storage engine for:
  - Massive scaling of concurrent writes and reads
  - High-volume OLTP with real-time performance
  - Built-in automatic failover and recovery
  - Online upgrades and scaling through the addition of nodes

MySQL Cluster has many strengths that make it an obvious choice for some workloads, but it is not always the best choice. When considering MySQL Cluster, you should consider the type of workload and data needs, as well as the environment in which it will operate.

Use MySQL Cluster if you are concerned about:

- Massive scalability
- High cost of down time
- The complexity of application-level sharding
- Building in high availability

Although InnoDB is not clustered, it might be a better choice than MySQL Cluster if you have:

- “Hot” data sets that are larger than you can fit in the cluster’s RAM
  - A 48-node MySQL Cluster with 512 GB RAM per node has a total of 24 TB RAM, which provides a typical in-memory working data size of under 12TB total with two nodes per node group.
  - InnoDB has a maximum size of 64TB per table.
- Complex transaction requirements such as selectable isolation levels, savepoints, and partial rollback
- Complex JOIN operations that cannot use adaptive query localization
- Large rows or full table scans
- Long-running transactions
- Geospatial indexes

## Notes

- For information about adaptive query localization, see the lesson titled “Optimizing MySQL Cluster Performance.”
- For additional information, see the *MySQL Cluster Evaluation Guide* at:  
[http://mysql.com/why-mysql/white-papers/mysql\\_cluster\\_eval\\_guide.php](http://mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php)

# Hardware and Software Requirements

MySQL Cluster has modest hardware and software requirements:

- Supported platforms include a broad range of operating systems and architectures:
  - Linux, Oracle Solaris, and Microsoft Windows
  - x86, x86-64, SPARC 32-bit and 64-bit
- Hardware requirements are similarly modest:
  - RAM: 1 GB suggested minimum
    - Benefits from large amounts of RAM
    - Live data storage handled in RAM
    - Can run on much less, for example when experimenting and testing in a lab environment
  - Hard drive: At least 3 GB

MySQL Cluster is designed to run on commodity hardware and has no requirements beyond those of a basic installation of the operating system.

In addition to the basic requirements outlined in the slide, MySQL Cluster benefits from additional memory. Because it predominantly uses an in-memory architecture, MySQL Cluster stores live data in RAM. As a result, large databases require large quantities of RAM, spread across multiple hosts.

For large volumes of data, you need a large hard drive to store cluster logs, checkpoints, and any disk data tables.

For details, see the following page:

<http://dev.mysql.com/doc/mysql/en/mysql-cluster-overview-requirements.html>

# Networking Requirements

- Network: 100 Mbps Ethernet (expected minimum)
  - Benefits from dedicated network and high-speed interconnects
- Cluster nodes connect to data nodes through dynamically allocated ports.
  - These might be blocked by:
    - Firewall on data nodes
    - Mandatory access control (MAC) systems on MySQL nodes
  - Specify a fixed port number with the `ServerPort` option.
  - Enable connectivity to the fixed port numbers in the firewall and other MAC systems (such as SELinux).
- Disable firewall and other access control to avoid latency.
  - Use separated networks where possible to ensure security.

You can improve the performance of networking between cluster nodes by using special networking configurations or hardware. These include TCP/IP direct connections, SCI (Scalable Coherent Interface) connections, or experimental (and unsupported) Shared Memory (SHM) connections.

In many clusters, the network bandwidth is a limiting factor. For example, a two-node cluster with four million updates per second of rows each containing ten 10-byte fields requires over 5 Gbps (including per-field and per-row overhead) to handle that traffic.

To minimize network latency, avoid using firewalls and access control between cluster hosts. Place cluster hosts on a dedicated network to ensure security. For example, configure application servers with one network interface visible on the user-facing network and with another interface dedicated to a physically separate cluster network. Disable port-forwarding between the interfaces.

On networks where firewalls or other access control systems are necessary, use fixed ports on data nodes, as described in the slide titled “Data Node: Global Configuration” in this lesson. Ensure that the port you choose is unique on each host.

**Note:** For information about configuring software firewalls and SELinux, see the lesson titled “Securing MySQL Cluster.”

# MySQL Cluster Distributions

The MySQL Cluster software distribution is available for:

- Operating systems and architectures
  - Enterprise Linux and general Linux (x86, x86\_64)
  - Oracle Solaris (x86/32 bit, x86\_64, SPARC 32-bit and 64-bit)
  - Microsoft Windows (x86, x86\_64)
  - Mac OS X (x86, x86\_64)
  - FreeBSD (x86\_64)
- Formats
  - Compressed archive: Extract and configure per-server.
    - `tar.gz`, `zip`
  - Native package installers: Install from package container.
    - RPM, DEB, PKG, MSI, DMG

MySQL Cluster downloads contain MySQL Server, compiled with support for the NDB storage engine. MySQL Server downloads do not contain or support MySQL Cluster.

MySQL Cluster software is available in many formats. Use the following MySQL Cluster native package installers for different platforms:

- RPM packages on Enterprise Linux systems such as Oracle Linux, Red Hat, or SuSE
- DEB format for Debian and Debian-derived systems such as Ubuntu
- PKG format for Oracle Solaris
- DMG format (containing a PKG installer) for Mac OS X
- MSI format for Windows

Compressed “no-install” MySQL Cluster archives are available in `tar.gz` file format for the UNIX-derived platforms and `zip` file format for Windows.

You can download the MySQL Cluster GPL edition source code in RPM format, or as a zip or `tar.gz` file.

You can get a distribution of MySQL Cluster bundled with MySQL Cluster Manager. For more information, see the lesson titled “MySQL Cluster Manager.”

**Note:** For a complete list of supported platforms, see the following page:

<http://www.mysql.com/support/supportedplatforms/cluster.html>

# Downloading MySQL Cluster

Download MySQL Cluster based on the edition:

- Community (GPL)
  - MySQL Downloads: <http://dev.mysql.com/downloads/cluster/>
  - MySQL Workbench Community Edition
- Carrier-grade (Commercial)
  - Oracle Software Delivery Cloud: <https://edelivery.oracle.com/>
    - Updates available from <https://support.oracle.com>
  - Includes MySQL Server Enterprise Edition
  - MySQL Cluster Manager is a separate download,
  - You can download other Enterprise Edition components:
    - MySQL Enterprise Monitor
    - MySQL Enterprise Backup
    - MySQL Workbench SE

To download MySQL Cluster from the Oracle Software Delivery Cloud or the My Oracle Support site (<https://support.oracle.com>), you must have an Oracle account. You already have one of these if you are an Oracle customer. Alternatively, you can register an account at no cost, which gives you access to trial versions of commercial software.

After you log on to the Oracle Software Delivery Cloud and agree to the terms, select the MySQL Database product pack and your system architecture to search for available MySQL Cluster downloads.

# Installing from RPM Packages

Use RPM packages when you install on Oracle Linux or other Enterprise Linux systems.

- Different RPMs for different components:
  - Server, client, development files, test packages, shared libraries, and embedded server
- Server RPM contains:
  - Daemon and utility binaries
  - MySQL Cluster Auto-Installer
  - Configuration files
  - Shared libraries and API adapters
  - Documentation
  - Localization files: Character sets and language files
- Client RPM contains client binaries and manuals.

MySQL Cluster is available in RPM format. In general, you use the server and client packages; other packages are available for the purposes of compiling your own MySQL client applications, running the MySQL test suite, and so on.

The server RPM package contains the following components:

- Configuration files
  - Startup scripts: `mysqld_safe`, `mysql.server`, and `/etc/init.d/mysql`
  - System configuration: SELinux and logrotate
  - Default `my.cnf`
- Binaries: `/usr/bin` for utilities and `/usr/sbin` for daemons
- Libraries: Client libraries, plug-ins, and API adapters
- Documentation, character sets, and languages
- MySQL Cluster Auto-Installer

The client RPM contains MySQL client applications such as `mysql`, `mysqlbinlog`, `mysqladmin`, `mysql_config_editor`, `mysqldump`, `mysqlshow`, `mysqlslap`, and so on.

For a listing of the RPM packages that are available for both MySQL Server and MySQL Cluster, go to <http://dev.mysql.com/doc/mysql/en/linux-installation-rpm.html>.

## RPM Installation Tasks

When you install from an RPM package, the installation script performs additional tasks:

- Sets the value of `datadir` to `/var/lib/mysql` on new installations
  - For upgrades, uses the previous value of `datadir` by reading from the output of `my_print_defaults`
- Creates the `mysql` user and group
  - Grants file permissions on the data directory to that user
- Configures SELinux settings on MySQL directories
- Creates or extends the `RPM_UPGRADE_HISTORY` file and creates the `RPM_UPGRADE_MARKER` file in the data directory
  - Renames the latter file to `RPM_UPGRADE_MARKER-LAST` on completion of the upgrade process

The `my.cnf` file contains MySQL Server configuration information and is required by MySQL Cluster SQL nodes. The `datadir` setting in that file configures the MySQL data directory, which contains metadata for all database tables and (by default) data for all tables that are not clustered using MySQL Cluster.

If the installation process detects the `RPM_UPGRADE_MARKER` file during installation, it assumes that a previous upgrade failed, and then terminates the process.

# Installing on Windows

Use an MSI installer or zipped binaries.

- The MSI installation file has contents equivalent to those of the RPM file.
  - Includes the MySQL Cluster Auto-Installer, all MySQL Cluster executables, and a MySQL Server instance
  - Places files in appropriate locations for Windows installations
    - Adds files and folders to Program Files and user profile folders
    - Adds Start Menu shortcuts
- The zipped archive contains all MySQL Cluster and MySQL Server files in a single compressed distribution.
  - Does not perform automatic post-install configuration
  - Provides most flexibility

Use the MySQL Cluster MSI installer on Windows to place program and data files in the typical locations for a Windows application.

Use the zipped binaries—called the “no-install” release—to extract all required files so that you can install them separately to locations of your choosing.

The MySQL Cluster installer does not install additional MySQL applications such as MySQL Workbench or MySQL for Excel. Similarly, it does not configure services. This contrasts with the more general MySQL Installer, which is a MySQL Server package for Windows that contains a full suite of MySQL Server tools and services but does not contain any MySQL Cluster software or support. In environments where you need software provided by both packages, install both but use the `mysqld.exe` binary provided by the MySQL Cluster package to connect to the cluster.

For details about installing MySQL Cluster on Windows from a zipped binary release, see the following page:

<http://dev.mysql.com/doc/mysql/en/mysql-cluster-install-windows-binary.html>

# Installing Management Nodes

MySQL Cluster binary release:

- Contains executables and supporting files for all cluster node types and other features
- Includes MySQL Cluster Auto-Installer
- Includes MySQL Server instance as a SQL node
- Does not perform any post-install configuration
  - You can extract all files to a file server without installation and subsequently copy files to their eventual hosts.
  - You must then configure each host manually.
- Is distributed in a compressed archive format that is appropriate to the operating system
  - `.tar.gz` file for Linux, Oracle Solaris, Mac OS X FreeBSD
  - `.zip` file for Windows

Use a binary release to provide greater control over where you want to put files (for example, for installation on different node types).

Typical usage:

- Extract all files to an initial location (for example, a host designated as a SQL node).
- Copy appropriate executables and supporting files to other hosts according to their designated node types.
- Perform all required configuration at each node.

The following slides explain the installation process in more detail. Further details on configuring each node type is in the lesson titled “Configuring MySQL Cluster.”

# Installing Management Nodes

- Each cluster must have at least one management node.
  - Use two or more for redundancy.
  - The cluster continues to run if there are no management nodes, but no additional nodes can join.
- Each management node requires:
  - The `ndb_mgmd` binary
  - A central cluster configuration file
  - A connect string to other management nodes (if any exist)
  - Incoming TCP/IP access on port 1186
- Launch with:

```
ndb_mgmd -f /etc/config.ini --initial # first time
ndb_mgmd                                # subsequent times
```

The MySQL Cluster management daemon requires the `ndb_mgmd` binary. Copy it to a suitable location on each management node host (for example, `/usr/sbin`).

Each management node also hosts a copy of the global configuration file for distribution to other nodes when they start. By convention, this file is named `config.ini`.

Management nodes listen on TCP port 1186 by default. Change this behavior by specifying a different `PortNumber` setting for each management node in the `config.ini` file. If a management node host has a firewall, ensure that it allows incoming connections from other cluster hosts on that port.

The first time you run each management node daemon in a cluster, provide the path to the configuration file with the `-f` option together with the `--initial` option, which instructs the management daemon to create the initial cached configuration in a configuration directory, which must exist before you launch the daemon.

If you have multiple management nodes in your cluster, provide a connect string with the `-c` option or in a `my.cnf` file that identifies other management nodes, as in the following example:

```
[ndb_mgmd]
config-file=/etc/config.ini
connect-string=mgmhost2
```

# Management Node: Global Configuration

- Each management node must have a copy of the shared global configuration file.
  - When a management node first reads the file, it caches the configuration in a directory.
  - When management nodes start, they read the configuration from other management nodes if available. Otherwise, they read them from the configuration cache directory.
- Minimal example:

```
[ndb_mgmd]
Hostname=mgmhost
[ndbd]
Hostname=datahost1
[ndbd]
Hostname=datahost2
[mysqld]
```

The default configuration cache directory for compressed archive installations is /usr/local/mysql/mysql-cluster, and for RPM installations, it is /usr/mysql-cluster. Override this with the --config-dir option.

Example:

```
ndb_mgmd -f /etc/config.ini --initial --config-dir=/etc/clusterconf
```

On subsequent invocations, each management daemon ignores the configuration file and reads its configuration from the cache directory. This means you have to provide only the location of the configuration cache directory. If you use the default configuration directory, you can invoke the management daemon with no parameters.

Management nodes store their logs and other working files in the binary's directory by default. Override this with the DataDir option.

**Note:** The minimal configuration shown in the slide is enough to start a basic cluster. However, a real-world cluster needs further configuration, as described in the lesson titled "Configuring MySQL Cluster."

# Installing Data Nodes

Each data node requires a data node daemon executable.

- Two available executables
  - **ndbd**: For single-threaded operation
  - **ndbmtd**: For multi-threaded operation on hosts with multiple available CPU cores
- Specify the host name of the management daemon in the /etc/my.cnf file:

```
[mysql_cluster]
ndb-connectstring=mgmhost
```

- Alternatively, you can specify the location of the management daemon when you launch the data node:

```
ndbmtd          # Management node in my.cnf
ndbmtd -c mgmhost # ...or at the command line
```

On data node hosts, copy the data node daemon executable from the extracted binary archive to a suitable location on the data node host (for example, /usr/sbin). Alternatively, you can install MySQL Cluster from a native package installer to place the daemon in the correct location.

All cluster node types read the [mysql\_cluster] section of the my.cnf file that is located on their host. In the example shown in the slide, all cluster nodes on that host—data nodes, API nodes, and even management nodes—read the ndb-connectstring value.

## Data Node: Global Configuration

Most configuration of data nodes is in the global config.ini file. Data nodes:

- Read settings from the [ndbd] and [ndbd default] sections
  - These sections are read by ndbd and ndbmtld processes.
- Store data and log files in the process working directory
  - Use the DataDir option to override this.
- Operate in single-threaded mode by default
  - ndbd data nodes are always single-threaded.
  - Configure multi-threaded mode for each ndbmtld data node:
    - ThreadConfig: Allows complex mapping of thread types to CPU cores
    - MaxNoOfExecutionThreads: Up to 36 execution threads

To configure settings for only the data nodes, place settings in the [ndbd] section. These settings apply to both single-threaded (ndbd) and multi-threaded (ndbmtld) data nodes. The [ndbd default] section contains settings that are read by all data nodes.

For example, to use a fixed rather than a dynamically allocated TCP port for the data node, use a configuration such as the following:

```
[ndbd]
Hostname=datahost1
ServerPort=55501
```

By default, data nodes store log files, data files, and other files in the process working directory. The DataDir option specifies the path to log files. The FileSystemPath option specifies the location of data files and metadata, as well as related logs, and by default it is the same as the DataDir. The BackupDataDir option also defaults to DataDir and specifies the location of backups created with the management client.

**Note:** For additional information about these and other options for configuring data nodes, see the lesson titled “Configuring MySQL Cluster.”

## Installing API Nodes

API nodes are the clients of MySQL Cluster and provide the interface through which users access data.

- API nodes host applications that connect to MySQL Cluster using NDB API or some other connector.
  - Examples: Java applications using ClusterJPA; C or C++ applications using the NDB API and compiled against NDB libraries
- Each application's configuration is specific to that application.
  - Cluster connectivity is part of the NDB API, and connection parameters are part of the application.
  - This includes MySQL Server instances:
    - The NDB storage engine must be enabled, and the instance must be connected to a management node.

Every application that connects to MySQL Cluster has its own requirements and configuration, as in the following examples:

- A Java application connecting to MySQL Cluster requires the application itself, along with a Java Runtime Environment and all required connectors such as ClusterJ or ClusterJPA.
- A NodeJS application requires NodeJS, along with the MySQL Connector for JavaScript, contained in the `share/node.js` directory in the extracted archive.

API nodes, including MySQL Server nodes, require network connectivity with data nodes. Ensure that firewall and access control systems permit outbound connections on dynamically allocated ports. Alternatively, use fixed ports (as described in the preceding slide).

# Installing and Configuring SQL Nodes

- Install MySQL Server SQL nodes as you install any other MySQL Server instance.
  - RPM installations perform post-installation user and permission configuration.
  - You must perform this configuration manually if you install from a compressed archive.
- Enable the NDB storage engine, connect string, and node ID in the SQL node's `my.cnf` file:

```
[mysqld]
# standard MySQL Server options
...
ndbcluster
ndb-nodeid=nodeid
ndb-connectstring=mgmhost
```

A SQL node is a special example of an API node. It uses an instance of MySQL Server with the NDB storage engine as an API node. Standard versions of MySQL Server do not include support for the NDB storage engine. MySQL Cluster binary packages include versions of MySQL Server that are built with support for that engine.

Note that RPM installations of MySQL Server create the `mysql` user and group, and they configure file system permissions and startup scripts so that this user launches the `mysqld` process and has full permissions to access the MySQL data directory. If you install from a compressed archive, you must perform such configuration manually.

In a cluster, the `mysqld` process accepts incoming connections on TCP port 3306 and sends outbound requests to management nodes on port 1186 and data nodes on their dynamically allocated or fixed ports. Ensure that any firewalls and access control systems allow such network communication.

The SQL node's configuration file must contain options enabling the NDB storage engine and the location of the cluster's management node, as well as the SQL node's node ID. Other API node options are in the cluster's global `config.ini` file and are managed by the management nodes.

# Starting a Cluster

You can administer the cluster from any client workstation.

- Use the `ndb_mgm` cluster management client to connect to the management node from any workstation.
  - Specify the location of the management daemon in the workstation's `/etc/my.cnf` file:

```
[ndb_mgm]
  ndb-connectstring=mgmhost
```

- Launch at the command line:

```
ndb_mgm          # with my.cnf settings
ndb_mgm -c mgmhost # without my.cnf settings
```

- Connect to a SQL node with MySQL Server clients:
  - Use standard MySQL clients such as `mysql`, `mysqladmin`, `mysqldump`, `mysqlslap`, `mysqlshow`, and so on.

You do not need to perform a complete installation of MySQL Cluster or MySQL Server on your workstation to administer a cluster.

Use the `ndb_mgm` cluster management client to perform administration tasks such as displaying the cluster's status, stopping specific nodes, backing up the cluster, and enabling cluster logging. The logging configuration commands of the cluster management client are covered in more detail in the lesson titled "Monitoring MySQL Cluster".

To create and configure clustered tables, use standard MySQL clients such as `mysql`, `mysqladmin`, or MySQL Workbench to connect to a SQL node that has the NDB storage engine enabled.

To back up both clustered and nonclustered tables, use `mysqldump` connected to such a SQL node. Note that `mysqldump` does not perform an online backup, so it cannot guarantee a consistent backup from an active cluster. The backup feature of the `ndb_mgm` client performs an online backup of clustered tables.

# Starting a Cluster

- The management node manages the configuration of all cluster nodes in its config.ini file.
  - At a minimum, management and data nodes require a Hostname parameter.
    - Directly, or indirectly via ExecuteOnComputer
- Typical clusters also define NoOfReplicas.
  - The default—and largest supported—value is 2.
  - A value of 1 does not provide redundancy.
- Other nodes require a connect string in the node's my.cnf file referring to the management nodes:

```
ndb-connectstring='mgmhost1,mgmhost2'
```

- SQL nodes also need to enable the NDB storage engine:

```
ndbcluster
```

When NoOfReplicas is 1, each data shard (partition) is hosted on a single data node. To enable high availability within the cluster, set NoOfReplicas to a value of at least 2.

In the config.ini file, the ExecuteOnComputer option of each node takes a numeric parameter. The number identifies the Id value of the [computer] section that contains the Hostname of the host on which the node runs. As a result, you can specify the host on which a node runs directly with Hostname in that node's section, or indirectly by providing an Id.

The following two configurations are equivalent:

```
1 [ndb_mgmd]
  Hostname=mgmhost

2 [ndb_mgmd]
  ExecuteOnComputer=4
  ...
  [computer]
  Id=4
  Hostname=mgmhost
```

# Starting a Cluster

To start a cluster, perform the following steps:

- 1. Start management nodes.**
  - Use `--initial` and identify the `config.ini` file when you first start a management node.
  - Ensure that each management node is made visible to other nodes in their connect strings, including other management nodes.
- 2. Start all data nodes.**
  - (Optional) Specify their node IDs in the connect string or with the `--ndb-nodeid` option.
- 3. Start API nodes to enable the application.**
  - To start MySQL nodes, use the startup scripts provided with the installer.

Management nodes assign node IDs to data nodes and API nodes automatically unless you specify node IDs explicitly, which you can do either in the connect string or with the `--ndb-nodeid` option. Data nodes can have IDs in the range 1–48. To accommodate the largest possible number of data nodes in very large clusters, specify node IDs greater than 48 for management and API nodes.

Follow MySQL Server best practices when using SQL nodes. For example, use the `mysqld_safe` or `mysql.server` service startup scripts that install with the RPM distribution, and use helper commands such as the following example used in Oracle Linux:

```
service mysql start
```

Data nodes cannot start until all management nodes have connected, and API nodes cannot start until all data nodes have connected. Use the `--nowait-nodes` option to start each data or management node without waiting on specified nodes of the same type.

For example, consider a cluster with data nodes 1–8. Assuming that you want to start only nodes 1–3, 6, and 7, use the following command on those hosts to start each of those nodes as node *N*, allowing API nodes to connect if all data nodes except 4, 5, and 8 have started:

```
ndbmtd --ndb-nodeid N --nowait-nodes=4,5,8
```

Use the cluster management client to view a running cluster.

- Run the client by using the `ndb_mgm` utility.
  - Provide the host name of a management node either in a connect string in `my.cnf` or with the `-c` option:

```
ndb_mgm -c mgmhost
```

- Use the `SHOW` command to view the cluster's status:

```
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 2 node(s)
id=1      @192.168.56.211  (mysql-5.6.17 ndb-7.3.5, Nodegroup: 0, *)
id=2      @192.168.56.212  (mysql-5.6.17 ndb-7.3.5, Nodegroup: 0)

[ndb_mgmd(MGM)] 2 node(s)
id=50     @192.168.56.215  (mysql-5.6.17 ndb-7.3.5)
id=51     @192.168.56.214  (mysql-5.6.17 ndb-7.3.5)

[mysqld(API)] 1 node(s)
id=100    @192.168.56.215  (mysql-5.6.17 ndb-7.3.5)
```

When you launch `ndb_mgm` with no parameters, it looks for a connect string in its configuration files. If it cannot find one, it uses `localhost` by default. This means you can easily run the management client on the same host as a management node. For this reason, the documentation recommends that you install `ndb_mgm` on all hosts that run `ndb_mgmd`.

You can run a single `ndb_mgm` command at the command prompt with the `-e` option.

Example:

```
# ndb_mgm -e 'SHOW'
Connected to Management Server at: host1:1186
Cluster Configuration
-----
[ndbd(NDB)] 2 node(s)
id=1 @192.168.100.201  (mysql-5.6.17 ndb-7.3.5, Nodegroup: 0, *)
id=2 @192.168.100.202  (mysql-5.6.17 ndb-7.3.5, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=49@192.168.100.201  (mysql-5.6.17 ndb-7.3.5)

[mysqld(API)] 1 node(s)
id=50@192.168.100.201  (mysql-5.6.17 ndb-7.3.5)
```

## Starting and Stopping Nodes with the Cluster Management Client

- Stop the whole cluster with the SHUTDOWN command.
  - This command stops data nodes and management nodes.
- Stop a single data node with the *nodeid* STOP command.
  - Stop all data nodes with the ALL STOP command.
- Start a single data node with the *nodeid* START command.
  - Start all data nodes with the ALL START command.
  - You must start the ndbd or ndbmtd process with the -n option.
- Restart a data node with the *nodeid* RESTART command.
  - Provide the -n option to perform a partial start so that you can use *nodeid* START at a later time.

The -n (or --nostart) option starts the communications components in the data node process, but does not start normal processing. The data node waits until the management node instructs it to start, which happens when you use the *nodeid* START command in the *ndb\_mgm* client. You can also provide the -n option when you execute *nodeid* RESTART.

The *nodeid* RESTART command does not stop a data node if it is the last remaining data node in a node group. You can override this check with the -f option. If stopping a data node with this option results in an incomplete cluster, the command restarts all data nodes in the cluster.



