MySQL®

Notes for Professionals

100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official MySQL® group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

Contents

About 1
Chapter 1: Getting started with MySQL 2
Section 1.1: Getting Started 2
Section 1.2: Information Schema Examples 6
Chapter 2: Data Types 7
Section 2.1: CHAR(n) 7
Section 2.2: DATE, DATETIME, TIMESTAMP, YEAR, and TIME 7
Section 2.3: VARCHAR(255) -- or not 8
Section 2.4: INT as AUTO_INCREMENT 8
Section 2.5: Others 8
Section 2.6: Implicit / automatic casting 9
Section 2.7: Introduction (numeric) 9
Section 2.8: Integer Types 10
Section 2.9: Fixed Point Types 10
Section 2.10: Floating Point Types 10
Section 2.11: Bit Value Type 11
Chapter 3: SELECT 12
Section 3.1: SELECT with DISTINCT 12
Section 3.2: SELECT all columns (*) 12
Section 3.3: SELECT by column name 13
Section 3.4: SELECT with LIKE (%) 13
Section 3.5: SELECT with CASE or IF 15
Section 3.6: SELECT with Alias (AS) 15
Section 3.7: SELECT with a LIMIT clause 16
Section 3.8: SELECT with BETWEEN 16
Section 3.9: SELECT with WHERE 18
Section 3.10: SELECT with LIKE(_) 18
Section 3.11: SELECT with date range 19
Chapter 4: Backticks 20
Section 4.1: Backticks usage 20
Chapter 5: NULL 21
Section 5.1: Uses for NULL 21
Section 5.2: Testing NULLs 21
Chapter 6: Limit and Offset 22
Section 6.1: Limit and Offset relationship 22
Chapter 7: Creating databases 24
Section 7.1: Create database, users, and grants 24
Section 7.2: Creating and Selecting a Database 26
Section 7.3: MyDatabase 26
Section 7.4: System Databases 27
Chapter 8: Using Variables 28
Section 8.1: Setting Variables 28
Section 8.2: Row Number and Group By using variables in Select Statement 29
Chapter 9: Comment MySQL 31
Section 9.1: Adding comments 31
Section 9.2: Commenting table definitions 31

Chapter 10: INSERT 32
Section 10.1: INSERT, ON DUPLICATE KEY UPDATE 32
Section 10.2: Inserting multiple rows 32
Section 10.3: Basic Insert 33
Section 10.4: INSERT with AUTO_INCREMENT + LAST_INSERT_ID() 33
Section 10.5: INSERT SELECT (Inserting data from another Table) 35
Section 10.6: Lost AUTO_INCREMENT ids 35
Chapter 11: DELETE 37
Section 11.1: Multi-Table Deletes 37
Section 11.2: DELETE vs TRUNCATE 39
Section 11.3: Multi-table DELETE 39
Section 11.4: Basic delete 39
Section 11.5: Delete with Where clause 39
Section 11.6: Delete all rows from a table 39
Section 11.7: LIMITing deletes 40
Chapter 12: UPDATE 41
Section 12.1: Update with Join Pattern 41
Section 12.2: Basic Update 41
Section 12.3: Bulk UPDATE 42
Section 12.4: UPDATE with ORDER BY and LIMIT 42
Section 12.5: Multiple Table UPDATE 42
Chapter 13: ORDER BY 44
Section 13.1: Contexts 44
Section 13.2: Basic 44
Section 13.3: ASCending / DESCending 44
Section 13.4: Some tricks 44
Chapter 14: Group By 46
Section 14.1: GROUP BY using HAVING 46
Section 14.2: Group By using Group Concat 46
Section 14.3: Group By Using MIN function 46
Section 14.4: GROUP BY with AGGREGATE functions 47
Chapter 15: Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause 50
Section 15.1: Misusing GROUP BY to return unpredictable results: Murphy's Law 50
Section 15.2: Misusing GROUP BY with SELECT *, and how to fix it 50
Section 15.3: ANY_VALUE() 51
Section 15.4: Using and misusing GROUP BY 51
Chapter 16: Joins 53
Section 16.1: Joins visualized 53
Section 16.2: JOIN with subquery ("Derived" table) 53
Section 16.3: Full Outer Join 54
Section 16.4: Retrieve customers with orders -- variations on a theme 55
Section 16.5: Joining Examples 56
Chapter 17: JOINS: Join 3 table with the same name of id. 57
Section 17.1: Join 3 tables on a column with the same name 57
Chapter 18: UNION 58
Section 18.1: Combining SELECT statements with UNION 58
Section 18.2: Combining data with different columns 58
Section 18.3: ORDER BY 58

Section 18.4: Pagination via OFFSET ... 58
Section 18.5: Combining and merging data on different MySQL tables with the same columns into unique rows and running query ... 59
Section 18.6: UNION ALL and UNION ... 59
Chapter 19: Arithmetic ... 60
Section 19.1: Arithmetic Operators ... 60
Section 19.2: Mathematical Constants ... 60
Section 19.3: Trigonometry (SIN, COS) ... 60
Section 19.4: Rounding (ROUND, FLOOR, CEIL) ... 62
Section 19.5: Raise a number to a power (POW) ... 62
Section 19.6: Square Root (SQRT) ... 63
Section 19.7: Random Numbers (RAND) ... 63
Section 19.8: Absolute Value and Sign (ABS, SIGN) ... 63
Chapter 20: String operations ... 65
Section 20.1: LENGTH() ... 66
Section 20.2: CHAR_LENGTH() ... 66
Section 20.3: HEX(str) ... 66
Section 20.4: SUBSTRING() ... 66
Section 20.5: UPPER() / UCASE() ... 67
Section 20.6: STR_TO_DATE - Convert string to date ... 67
Section 20.7: LOWER() / LCASE() ... 67
Section 20.8: REPLACE() ... 67
Section 20.9: Find element in comma separated list ... 67
Chapter 21: Date and Time Operations ... 69
Section 21.1: Date arithmetic ... 69
Section 21.2: SYSDATE(), NOW(), CURDATE() ... 69
Section 21.3: Testing against a date range ... 70
Section 21.4: Extract Date from Given Date or DateTime Expression ... 70
Section 21.5: Using an index for a date and time lookup ... 70
Section 21.6: Now() ... 71
Chapter 22: Handling Time Zones ... 72
Section 22.1: Retrieve the current date and time in a particular time zone ... 72
Section 22.2: Convert a stored `DATE` or `DATETIME` value to another time zone ... 72
Section 22.3: Retrieve stored `TIMESTAMP` values in a particular time zone ... 72
Section 22.4: What is my server's local time zone setting? ... 72
Section 22.5: What time_zone values are available in my server? ... 73
Chapter 23: Regular Expressions ... 74
Section 23.1: REGEXP / RLIKE ... 74
Chapter 24: VIEW ... 76
Section 24.1: Create a View ... 76
Section 24.2: A view from two tables ... 77
Section 24.3: DROPPING A VIEW ... 77
Section 24.4: Updating a table via a VIEW ... 77
Chapter 25: Table Creation ... 78
Section 25.1: Table creation with Primary Key ... 78
Section 25.2: Basic table creation ... 79
Section 25.3: Table creation with Foreign Key ... 79
Section 25.4: Show Table Structure ... 80
Section 25.5: Cloning an existing table ... 81

Section 25.6: Table Create With TimeStamp Column To Show Last Update ... 81
Section 25.7: CREATE TABLE FROM SELECT ... 81
Chapter 26: ALTER TABLE ... 83
Section 26.1: Changing storage engine; rebuild table; change file_per_table ... 83
Section 26.2: ALTER COLUMN OF TABLE ... 83
Section 26.3: Change auto-increment value ... 83
Section 26.4: Renaming a MySQL table ... 83
Section 26.5: ALTER table add INDEX ... 84
Section 26.6: Changing the type of a primary key column ... 84
Section 26.7: Change column definition ... 84
Section 26.8: Renaming a MySQL database ... 84
Section 26.9: Swapping the names of two MySQL databases ... 85
Section 26.10: Renaming a column in a MySQL table ... 85
Chapter 27: Drop Table ... 87
Section 27.1: Drop Table ... 87
Section 27.2: Drop tables from database ... 87
Chapter 28: MySQL LOCK TABLE ... 88
Section 28.1: Row Level Locking ... 88
Section 28.2: Mysql Locks ... 89
Chapter 29: Error codes ... 91
Section 29.1: Error code 1064: Syntax error ... 91
Section 29.2: Error code 1175: Safe Update ... 91
Section 29.3: Error code 1215: Cannot add foreign key constraint ... 91
Section 29.4: 1067, 1292, 1366, 1411 - Bad Value for number, date, default, etc ... 93
Section 29.5: 1045 Access denied ... 93
Section 29.6: 1236 "impossible position" in Replication ... 93
Section 29.7: 2002, 2003 Cannot connect ... 94
Section 29.8: 126, 127, 134, 144, 145 ... 94
Section 29.9: 139 ... 94
Section 29.10: 1366 ... 94
Section 29.11: 126, 1054, 1146, 1062, 24 ... 95
Chapter 30: Stored routines (procedures and functions) ... 97
Section 30.1: Stored procedure with IN, OUT, INOUT parameters ... 97
Section 30.2: Create a Function ... 98
Section 30.3: Cursors ... 99
Section 30.4: Multiple ResultSets ... 100
Section 30.5: Create a function ... 100
Chapter 31: Indexes and Keys ... 102
Section 31.1: Create index ... 102
Section 31.2: Create unique index ... 102
Section 31.3: AUTO_INCREMENT key ... 102
Section 31.4: Create composite index ... 102
Section 31.5: Drop index ... 103
Chapter 32: Full-Text search ... 104
Section 32.1: Simple FULLTEXT search ... 104
Section 32.2: Simple BOOLEAN search ... 104
Section 32.3: Multi-column FULLTEXT search ... 104
Chapter 33: PREPARE Statements ... 106
Section 33.1: PREPARE, EXECUTE and DEALLOCATE PREPARE Statements ... 106

Section 33.2: Alter table with add column 106
Chapter 34: JSON 107
Section 34.1: Create simple table with a primary key and JSON field 107
Section 34.2: Insert a simple JSON 107
Section 34.3: Updating a JSON field 107
Section 34.4: Insert mixed data into a JSON field 108
Section 34.5: CAST data to JSON type 108
Section 34.6: Create Json Object and Array 108
Chapter 35: Extract values from JSON type 109
Section 35.1: Read JSON Array value 109
Section 35.2: JSON Extract Operators 109
Chapter 36: MySQL Admin 111
Section 36.1: Atomic RENAME & Table Reload 111
Section 36.2: Change root password 111
Section 36.3: Drop database 111
Chapter 37: TRIGGERS 112
Section 37.1: Basic Trigger 112
Section 37.2: Types of triggers 112
Chapter 38: Configuration and tuning 114
Section 38.1: InnoDB performance 114
Section 38.2: Parameter to allow huge data to insert 114
Section 38.3: Increase the string limit for group_concat 114
Section 38.4: Minimal InnoDB configuration 114
Section 38.5: Secure MySQL encryption 115
Chapter 39: Events 116
Section 39.1: Create an Event 116
Chapter 40: ENUM 119
Section 40.1: Why ENUM? 119
Section 40.2: VARCHAR as an alternative 119
Section 40.3: Adding a new option 119
Section 40.4: NULL vs NOT NULL 119
Chapter 41: Install Mysql container with Docker-Compose 121
Section 41.1: Simple example with docker-compose 121
Chapter 42: Character Sets and Collations 122
Section 42.1: Which CHARACTER SET and COLLATION? 122
Section 42.2: Setting character sets on tables and fields 122
Section 42.3: Declaration 122
Section 42.4: Connection 123
Chapter 43: MyISAM Engine 124
Section 43.1: ENGINE=MyISAM 124
Chapter 44: Converting from MyISAM to InnoDB 125
Section 44.1: Basic conversion 125
Section 44.2: Converting All Tables in one Database 125
Chapter 45: Transaction 126
Section 45.1: Start Transaction 126
Section 45.2: COMMIT , ROLLBACK and AUTOCOMMIT 127
Section 45.3: Transaction using JDBC Driver 129
Chapter 46: Log files 132

Section 46.1: Slow Query Log 132
Section 46.2: A List 132
Section 46.3: General Query Log 133
Section 46.4: Error Log 134
Chapter 47: Clustering 136
Section 47.1: Disambiguation 136
Chapter 48: Partitioning 137
Section 48.1: RANGE Partitioning 137
Section 48.2: LIST Partitioning 137
Section 48.3: HASH Partitioning 138
Chapter 49: Replication 139
Section 49.1: Master - Slave Replication Setup 139
Section 49.2: Replication Errors 141
Chapter 50: Backup using mysqldump 143
Section 50.1: Specifying username and password 143
Section 50.2: Creating a backup of a database or table 143
Section 50.3: Restoring a backup of a database or table 144
Section 50.4: Tranferring data from one MySQL server to another 144
Section 50.5: mysqldump from a remote server with compression 145
Section 50.6: restore a gzipped mysqldump file without uncompressing 145
Section 50.7: Backup database with stored procedures and functions 145
Section 50.8: Backup direct to Amazon S3 with compression 145
Chapter 51: mysqlimport 146
Section 51.1: Basic usage 146
Section 51.2: Using a custom field-delimiter 146
Section 51.3: Using a custom row-delimiter 146
Section 51.4: Handling duplicate keys 146
Section 51.5: Conditional import 147
Section 51.6: Import a standard csv 147
Chapter 52: LOAD DATA INFILE 148
Section 52.1: using LOAD DATA INFILE to load large amount of data to database 148
Section 52.2: Load data with duplicates 149
Section 52.3: Import a CSV file into a MySQL table 149
Chapter 53: MySQL Unions 150
Section 53.1: Union operator 150
Section 53.2: Union ALL 150
Section 53.3: UNION ALL With WHERE 151
Chapter 54: MySQL client 152
Section 54.1: Base login 152
Section 54.2: Execute commands 152
Chapter 55: Temporary Tables 154
Section 55.1: Create Temporary Table 154
Section 55.2: Drop Temporary Table 154
Chapter 56: Customize PS1 155
Section 56.1: Customize the MySQL PS1 with current database 155
Section 56.2: Custom PS1 via MySQL configuration file 155
Chapter 57: Dealing with sparse or missing data 156
Section 57.1: Working with columns containg NULL values 156

Chapter 58: Connecting with UTF-8 Using Various Programming language. 159
Section 58.1: Python 159
Section 58.2: PHP 159
Chapter 59: Time with subsecond precision 160
Section 59.1: Get the current time with millisecond precision 160
Section 59.2: Get the current time in a form that looks like a Javascript timestamp 160
Section 59.3: Create a table with columns to store sub-second time 160
Section 59.4: Convert a millisecond-precision date / time value to text 160
Section 59.5: Store a Javascript timestamp into a TIMESTAMP column 161
Chapter 60: One to Many 162
Section 60.1: Example Company Tables 162
Section 60.2: Get the Employees Managed by a Single Manager 162
Section 60.3: Get the Manager for a Single Employee 162
Chapter 61: Server Information 164
Section 61.1: SHOW VARIABLES example 164
Section 61.2: SHOW STATUS example 164
Chapter 62: SSL Connection Setup 166
Section 62.1: Setup for Debian-based systems 166
Section 62.2: Setup for CentOS7 / RHEL7 168
Chapter 63: Create New User 173
Section 63.1: Create a MySQL User 173
Section 63.2: Specify the password 173
Section 63.3: Create new user and grant all priviliges to schema 173
Section 63.4: Renaming user 173
Chapter 64: Security via GRANTs 174
Section 64.1: Best Practice 174
Section 64.2: Host (of user@host) 174
Chapter 65: Change Password 175
Section 65.1: Change MySQL root password in Linux 175
Section 65.2: Change MySQL root password in Windows 175
Section 65.3: Process 176
Chapter 66: Recover and reset the default root password for MySQL 5.7+ 177
Section 66.1: What happens when the initial start up of the server 177
Section 66.2: How to change the root password by using the default password 177
Section 66.3: reset root password when '' /var/run/mysqld' for UNIX socket file don't exists'' 177
Chapter 67: Recover from lost root password 180
Section 67.1: Set root password, enable root user for socket and http access 180
Chapter 68: MySQL Performance Tips 181
Section 68.1: Building a composite index 181
Section 68.2: Optimizing Storage Layout for InnoDB Tables 181
Chapter 69: Performance Tuning 183
Section 69.1: Don't hide in function 183
Section 69.2: OR 183
Section 69.3: Add the correct index 183
Section 69.4: Have an INDEX 184
Section 69.5: Subqueries 184
Section 69.6: JOIN + GROUP BY 184
Section 69.7: Set the cache correctly 185

Section 69.8: Negatives 185
Appendix A: Reserved Words 186
Section A.1: Errors due to reserved words 186
Credits 187
You may also like 190

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/MySQLBook

This *MySQL® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official MySQL® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with MySQL

Version	Release Date
1.0	1995-05-23
3.19	1996-12-01
3.20	1997-01-01
3.21	1998-10-01
3.22	1999-10-01
3.23	2001-01-22
4.0	2003-03-01
4.1	2004-10-01
5.0	2005-10-01
5.1	2008-11-27
5.5	2010-11-01
5.6	2013-02-01
5.7	2015-10-01

Section 1.1: Getting Started

Creating a database in MySQL

```
CREATE DATABASE mydb;
```

Return value:

> Query OK, 1 row affected (0.05 sec)

Using the created database `mydb`

```
USE mydb;
```

Return value:

> Database Changed

Creating a table in MySQL

```
CREATE TABLE mytable
(
  id              int unsigned NOT NULL auto_increment,
  username        varchar(100) NOT NULL,
  email           varchar(100) NOT NULL,
  PRIMARY KEY     (id)
);
```

`CREATE TABLE mytable` will create a new table called `mytable`.

`id int unsigned NOT NULL auto_increment` creates the `id` column, this type of field will assign a unique numeric ID to each record in the table (meaning that no two rows can have the same `id` in this case), MySQL will

automatically assign a new, unique value to the record's `id` field (starting with 1).

Return value:

> Query OK, 0 rows affected (0.10 sec)

Inserting a row into a MySQL table

```
INSERT INTO mytable ( username, email )
VALUES ( "myuser", "myuser@example.com" );
```

Example return value:

> Query OK, 1 row affected (0.06 sec)

The `varchar` a.k.a `strings` can be also be inserted using single quotes:

```
INSERT INTO mytable ( username, email )
VALUES ( 'username', 'username@example.com' );
```

Updating a row into a MySQL table

```
UPDATE mytable SET username="myuser" WHERE id=8
```

Example return value:

> Query OK, 1 row affected (0.06 sec)

The `int` value can be inserted in a query without quotes. Strings and Dates must be enclosed in single quote `'` or double quotes `"`.

Deleting a row into a MySQL table

```
DELETE FROM mytable WHERE id=8
```

Example return value:

> Query OK, 1 row affected (0.06 sec)

This will delete the row having `id` is 8.

Selecting rows based on conditions in MySQL

```
SELECT * FROM mytable WHERE username = "myuser";
```

Return value:

```
+----+----------+--------------------+
| id | username | email              |
```

```
+----+----------+---------------------+
|  1 | myuser   | myuser@example.com  |
+----+----------+---------------------+
```

> 1 row in set (0.00 sec)

Show list of existing databases

```
SHOW databases;
```

Return value:

```
+------------------+
| Databases        |
+------------------+
| information_schema|
| mydb             |
+------------------+
```

> 2 rows in set (0.00 sec)

You can think of "information_schema" as a "master database" that provides access to database metadata.

Show tables in an existing database

```
SHOW tables;
```

Return value:

```
+----------------+
| Tables_in_mydb |
+----------------+
| mytable        |
+----------------+
```

> 1 row in set (0.00 sec)

Show all the fields of a table

```
DESCRIBE databaseName.tableName;
```

or, if already using a database:

```
DESCRIBE tableName;
```

Return value:

```
+----------+---------------+-------+---------+------------------+-------+
```

```
| Field     | Type           | Null   | Key     | Default           | Extra |
+-----------+----------------+--------+---------+-------------------+-------+
| fieldname | fieldvaluetype | NO/YES | keytype | defaultfieldvalue |       |
+-----------+----------------+--------+---------+-------------------+-------+
```

`Extra` may contain `auto_increment` for example.

`Key` refers to the type of key that may affect the field. Primary (PRI), Unique (UNI) ...

> n row in set (0.00 sec)

Where n is the number of fields in the table.

Creating user

First, you need to create a user and then give the user permissions on certain databases/tables. While creating the user, you also need to specify where this user can connect from.

```
CREATE USER 'user'@'localhost' IDENTIFIED BY 'some_password';
```

Will create a user that can only connect on the local machine where the database is hosted.

```
CREATE USER 'user'@'%' IDENTIFIED BY 'some_password';
```

Will create a user that can connect from anywhere (except the local machine).

Example return value:

> Query OK, 0 rows affected (0.00 sec)

Adding privileges

Grant common, basic privileges to the user for all tables of the specified database:

```
GRANT SELECT, INSERT, UPDATE ON databaseName.* TO 'userName'@'localhost';
```

Grant all privileges to the user for all tables on all databases (attention with this):

```
GRANT ALL ON *.* TO 'userName'@'localhost' WITH GRANT OPTION;
```

As demonstrated above, `*.*` targets all databases and tables, `databaseName.*` targets all tables of the specific database. It is also possible to specify database and table like so `databaseName.tableName`.

`WITH GRANT OPTION` should be left out if the user need not be able to grant other users privileges.

Privileges can be **either**

```
ALL
```

or a combination of the following, each separated by a comma (non-exhaustive list).

```
SELECT
```

```
INSERT
UPDATE
DELETE
CREATE
DROP
```

Note

Generally, you should try to avoid using column or table names containing spaces or using reserved words in SQL. For example, it's best to avoid names like `table` or `first name`.

If you must use such names, put them between back-tick `` ` `` delimiters. For example:

```
CREATE TABLE `table`
(
   `first name` VARCHAR(30)
);
```

A query containing the back-tick delimiters on this table might be:

```
SELECT `first name` FROM `table` WHERE `first name` LIKE 'a%';
```

Section 1.2: Information Schema Examples

Processlist
This will show all active & sleeping queries in that order then by how long.

```
SELECT * FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

This is a bit more detail on time-frames as it is in seconds by default

```
SELECT ID, USER, HOST, DB, COMMAND,
TIME as time_seconds,
ROUND(TIME / 60, 2) as time_minutes,
ROUND(TIME / 60 / 60, 2) as time_hours,
STATE, INFO
FROM information_schema.PROCESSLIST ORDER BY INFO DESC, TIME DESC;
```

Stored Procedure Searching

Easily search thru all `Stored Procedures` for words and wildcards.

```
SELECT * FROM information_schema.ROUTINES WHERE ROUTINE_DEFINITION LIKE '%word%';
```

Chapter 2: Data Types

Section 2.1: CHAR(n)

`CHAR(n)` is a string of a *fixed* length of `n` *characters*. If it is `CHARACTER SET utf8mb4`, that means it occupies exactly `4*n` *bytes*, regardless of what text is in it.

Most use cases for `CHAR(n)` involve strings that contain English characters, hence should be `CHARACTER SET ascii`. (`latin1` will do just as good.)

```
country_code CHAR(2) CHARACTER SET ascii,
postal_code  CHAR(6) CHARACTER SET ascii,
uuid    CHAR(39) CHARACTER SET ascii,  -- more discussion elsewhere
```

Section 2.2: DATE, DATETIME, TIMESTAMP, YEAR, and TIME

The `DATE` datatype comprises the date but no time component. Its format is `'YYYY-MM-DD'` with a range of '1000-01-01' to '9999-12-31'.

The `DATETIME` type includes the time with a format of 'YYYY-MM-DD HH:MM:SS'. It has a range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

The `TIMESTAMP` type is an integer type comprising date and time with an effective range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.

The `YEAR` type represents a year and holds a range from 1901 to 2155.

The `TIME` type represents a time with a format of 'HH:MM:SS' and holds a range from '-838:59:59' to '838:59:59'.

Storage Requirements:

```
|-----------|--------------------|-----------------------------------------|
| Data Type | Before MySQL 5.6.4 | as of MySQL 5.6.4                       |
|-----------|--------------------|-----------------------------------------|
| YEAR      |      1 byte        |  1 byte                                 |
| DATE      |      3 bytes       |  3 bytes                                |
| TIME      |      3 bytes       |  3 bytes + fractional seconds storage   |
| DATETIME  |      8 bytes       |  5 bytes + fractional seconds storage   |
| TIMESTAMP |      4 bytes       |  4 bytes + fractional seconds storage   |
|-----------|--------------------|-----------------------------------------|
```

Fractional Seconds (as of Version 5.6.4):

```
|------------------------------|------------------|
| Fractional Seconds Precision | Storage Required |
|------------------------------|------------------|
|              0               |      0 bytes     |
|              1,2             |      1 byte      |
|              3,4             |      2 byte      |
|              5,6             |      3 byte      |
|------------------------------|------------------|
```

See the MySQL Manual Pages DATE, DATETIME, and TIMESTAMP Types, Data Type Storage Requirements, and Fractional Seconds in Time Values.

Section 2.3: VARCHAR(255) -- or not

Suggested max len

First, I will mention some common strings that are always hex, or otherwise limited to ASCII. For these, you should specify `CHARACTER SET ascii` (`latin1` is ok) so that it will not waste space:

```
UUID CHAR(36) CHARACTER SET ascii -- or pack into BINARY(16)
country_code CHAR(2) CHARACTER SET ascii
ip_address CHAR(39) CHARACTER SET ascii -- or pack into BINARY(16)
phone VARCHAR(20) CHARACTER SET ascii -- probably enough to handle extension
postal_code VARCHAR(20) CHARACTER SET ascii -- (not 'zip_code') (don't know the max

city VARCHAR(100) -- This Russian town needs 91:
    Poselok Uchebnogo Khozyaystva Srednego Professionalno-Tekhnicheskoye Uchilishche Nomer Odin
country VARCHAR(50) -- probably enough
name VARCHAR(64) -- probably adequate; more than some government agencies allow
```

Why not simply 255? There are two reasons to avoid the common practice of using (255) for everything.

- When a complex `SELECT` needs to create temporary table (for a subquery, `UNION`, `GROUP BY`, etc), the preferred choice is to use the `MEMORY` engine, which puts the data in RAM. But `VARCHAR`s are turned into `CHAR` in the process. This makes `VARCHAR(255) CHARACTER SET utf8mb4` take 1020 bytes. That can lead to needing to spill to disk, which is slower.
- In certain situations, InnoDB will look at the potential size of the columns in a table and decide that it will be too big, aborting a `CREATE TABLE`.

VARCHAR versus **TEXT**

Usage hints for `*TEXT`, `CHAR`, and `VARCHAR`, plus some Best Practice:

- Never use `TINYTEXT`.
- Almost never use `CHAR` -- it is fixed length; each character is the max length of the `CHARACTER SET` (eg, 4 bytes/character for utf8mb4).
- With `CHAR`, use `CHARACTER SET ascii` unless you know otherwise.
- `VARCHAR(n)` will truncate at n *characters*; `TEXT` will truncate at some number of *bytes*. (But, do you want truncation?)
- `*TEXT` *may* slow down complex `SELECT`s due to how temp tables are handled.

Section 2.4: INT as AUTO_INCREMENT

Any size of `INT` may be used for `AUTO_INCREMENT`. `UNSIGNED` is always appropriate.

Keep in mind that certain operations "burn" `AUTO_INCREMENT` ids. This could lead to an unexpected gap. Examples: `INSERT IGNORE` and `REPLACE`. They *may* preallocate an id *before* realizing that it won't be needed. This is expected behavior and by design in the InnoDB engine and should not discourage their use.

Section 2.5: Others

There is already a separate entry for "FLOAT, DOUBLE, and DECIMAL" and "ENUM". A single page on datatypes is likely to be unwieldy -- I suggest "Field types" (or should it be called "Datatypes"?) be an overview, then split into these topic pages:

- INTs
- FLOAT, DOUBLE, and DECIMAL

- Strings (CHARs, TEXT, etc)
- BINARY and BLOB
- DATETIME, TIMESTAMP, and friends
- ENUM and SET
- Spatial data
- JSON type (MySQL 5.7.8+)
- How to represent Money, and other common 'types' that need shoehorning into existing datatypes

Where appropriate, each topic page should include, in addition to syntax and examples:

- Considerations when ALTERing
- Size (bytes)
- Contrast with non-MySQL engines (low priority)
- Considerations when using the datatype in a PRIMARY KEY or secondary key
- other Best Practice
- other Performance issues

(I assume this "example" will self-distruct when my suggestions have been satisfied or vetoed.)

Section 2.6: Implicit / automatic casting

```
select '123' * 2;
```

To make the **multiplication** with 2 MySQL automatically converts the string `123` into a number.

Return value:

> 246

The conversion to a number starts from left to right. If the conversion is not possible the result is `0`

```
select '123ABC' * 2
```

Return value:

> 246

```
select 'ABC123' * 2
```

Return value:

> 0

Section 2.7: Introduction (numeric)

MySQL offers a number of different numeric types. These can be broken down into

Group	Types
Integer Types	`INTEGER`, `INT`, `SMALLINT`, `TINYINT`, `MEDIUMINT`, `BIGINT`

Fixed Point Types	`DECIMAL`, `NUMERIC`
Floating Point Types	`FLOAT`, `DOUBLE`
Bit Value Type	`BIT`

Section 2.8: Integer Types

Minimal unsigned value is always 0.

Type	Storage (Bytes)	Minimum Value (Signed)	Maximum Value (Signed)	Maximum Value (Unsigned)
`TINYINT`	1	-27 -128	27-1 127	28-1 255
`SMALLINT`	2	-215 -32,768	215-1 32,767	216-1 65,535
`MEDIUMINT`	3	-223 -8,388,608	223-1 8,388,607	224-1 16,777,215
`INT`	4	-231 -2,147,483,648	231-1 2,147,483,647	232-1 4,294,967,295
`BIGINT`	8	-263 -9,223,372,036,854,775,808	263-1 9,223,372,036,854,775,807	264-1 18,446,744,073,709,551,615

Section 2.9: Fixed Point Types

MySQL's `DECIMAL` and `NUMERIC` types store exact numeric data values. It is recommended to use these types to preserve exact precision, such as for money.

Decimal

These values are stored in binary format. In a column declaration, the precision and scale should be specified

Precision represents the number of significant digits that are stored for values.

Scale represents the number of digits stored *after* the decimal

```
salary DECIMAL(5,2)
```

5 represents the `precision` and 2 represents the `scale`. For this example, the range of values that can be stored in this column is `-999.99 to 999.99`

If the scale parameter is omitted, it defaults to 0

This data type can store up to 65 digits.

The number of bytes taken by `DECIMAL(M,N)` is *approximately* `M/2`.

Section 2.10: Floating Point Types

`FLOAT` and `DOUBLE` represent *approximate* data types.

Type	Storage	Precision	Range
FLOAT	4 bytes	23 significant bits / ~7 decimal digits	10^+/-38
DOUBLE	8 bytes	53 significant bits / ~16 decimal digits	10^+/-308

`REAL` is a synonym for `FLOAT`. `DOUBLE PRECISION` is a synonym for `DOUBLE`.

Although MySQL also permits (M,D) qualifier, do *not* use it. (M,D) means that values can be stored with up to M total digits, where D can be after the decimal. *Numbers will be rounded twice or truncated; this will cause more trouble than benefit.*

Because floating-point values are approximate and not stored as exact values, attempts to treat them as exact in comparisons may lead to problems. Note in particular that a `FLOAT` value rarely equals a `DOUBLE` value.

Section 2.11: Bit Value Type

The `BIT` type is useful for storing bit-field values. `BIT(M)` allows storage of up to M-bit values where M is in the range of `1 to 64`

You can also specify values with `bit value` notation.

```
b'111'      -> 7
b'10000000' -> 128
```

Sometimes it is handy to use 'shift' to construct a single-bit value, for example `(1 << 7)` for 128.

The maximum combined size of all BIT columns in an `NDB` table is 4096.

Chapter 3: SELECT

`SELECT` is used to retrieve rows selected from one or more tables.

Section 3.1: SELECT with DISTINCT

The `DISTINCT` clause after `SELECT` eliminates duplicate rows from the result set.

```
CREATE TABLE `car`
(    `car_id` INT UNSIGNED NOT NULL PRIMARY KEY,
    `name` VARCHAR(20),
    `price` DECIMAL(8,2)
);

INSERT INTO CAR (`car_id`, `name`, `price`) VALUES (1, 'Audi A1', '20000');
INSERT INTO CAR (`car_id`, `name`, `price`) VALUES (2, 'Audi A1', '15000');
INSERT INTO CAR (`car_id`, `name`, `price`) VALUES (3, 'Audi A2', '40000');
INSERT INTO CAR (`car_id`, `name`, `price`) VALUES (4, 'Audi A2', '40000');

SELECT DISTINCT `name`, `price` FROM CAR;
```

```
+---------+----------+
| name    | price    |
+---------+----------+
| Audi A1 | 20000.00 |
| Audi A1 | 15000.00 |
| Audi A2 | 40000.00 |
+---------+----------+
```

`DISTINCT` works across all columns to deliver the results, not individual columns. The latter is often a misconception of new SQL developers. In short, it is the distinctness at the row-level of the result set that matters, not distinctness at the column-level. To visualize this, look at "Audi A1" in the above result set.

For later versions of MySQL, `DISTINCT` has implications with its use alongside `ORDER BY`. The setting for `ONLY_FULL_GROUP_BY` comes into play as seen in the following MySQL Manual Page entitled MySQL Handling of GROUP BY.

Section 3.2: SELECT all columns (*)

Query

```
SELECT * FROM stack;
```

Result

```
+------+----------+----------+
| id   | username | password |
+------+----------+----------+
|    1 | admin    | admin    |
|    2 | stack    | stack    |
+------+----------+----------+
2 rows in set (0.00 sec)
```

You can select all columns from one table in a join by doing:

```
SELECT stack.* FROM stack JOIN Overflow ON stack.id = Overflow.id;
```

Best Practice Do not use * unless you are debugging or fetching the row(s) into associative arrays, otherwise schema changes (ADD/DROP/rearrange columns) can lead to nasty application errors. Also, if you give the list of columns you need in your result set, MySQL's query planner often can optimize the query.

Pros:

1. When you add/remove columns, you don't have to make changes where you did use `SELECT *`
2. It's shorter to write
3. You also see the answers, so can `SELECT *`-usage ever be justified?

Cons:

1. You are returning more data than you need. Say you add a VARBINARY column that contains 200k per row. You only need this data in one place for a single record - using `SELECT *` you can end up returning 2MB per 10 rows that you don't need
2. Explicit about what data is used
3. Specifying columns means you get an error when a column is removed
4. The query processor has to do some more work - figuring out what columns exist on the table (thanks @vinodadhikary)
5. You can find where a column is used more easily
6. You get all columns in joins if you use SELECT *
7. You can't safely use ordinal referencing (though using ordinal references for columns is bad practice in itself)
8. In complex queries with `TEXT` fields, the query may be slowed down by less-optimal temp table processing

Section 3.3: SELECT by column name

```
CREATE TABLE stack(
    id INT,
    username VARCHAR(30) NOT NULL,
    password VARCHAR(30) NOT NULL
);

INSERT INTO stack (`id`, `username`, `password`) VALUES (1, 'Foo', 'hiddenGem');
INSERT INTO stack (`id`, `username`, `password`) VALUES (2, 'Baa', 'verySecret');
```

Query

```
SELECT id FROM stack;
```

Result

```
+------+
| id   |
+------+
|    1 |
|    2 |
+------+
```

Section 3.4: SELECT with LIKE (%)

```
CREATE TABLE stack
```

```
(  id int AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(100) NOT NULL
);

INSERT stack(username) VALUES
('admin'),('k admin'),('adm'),('a adm b'),('b XadmY c'), ('adm now'), ('not here');
```

"adm" anywhere:

```
SELECT * FROM stack WHERE username LIKE "%adm%";
```

```
+----+-----------+
| id | username  |
+----+-----------+
|  1 | admin     |
|  2 | k admin   |
|  3 | adm       |
|  4 | a adm b   |
|  5 | b XadmY c |
|  6 | adm now   |
+----+-----------+
```

Begins with "adm":

```
SELECT * FROM stack WHERE username LIKE "adm%";
```

```
+----+----------+
| id | username |
+----+----------+
|  1 | admin    |
|  3 | adm      |
|  6 | adm now  |
+----+----------+
```

Ends with "adm":

```
SELECT * FROM stack WHERE username LIKE "%adm";
```

```
+----+----------+
| id | username |
+----+----------+
|  3 | adm      |
+----+----------+
```

Just as the `%` character in a `LIKE` clause matches any number of characters, the `_` character matches just one character. For example,

```
SELECT * FROM stack WHERE username LIKE "adm_n";
```

```
+----+----------+
| id | username |
+----+----------+
|  1 | admin    |
+----+----------+
```

Performance Notes If there is an index on `username`, then

- `LIKE 'adm'` performs the same as `= 'adm'
- `LIKE 'adm%` is a "range", similar to `BETWEEN..AND..` It can make good use of an index on the column.
- `LIKE '%adm'` (or any variant with a *leading* wildcard) cannot use any index. Therefore it will be slow. On tables with many rows, it is likely to be so slow it is useless.
- `RLIKE` (`REGEXP`) tends to be slower than `LIKE`, but has more capabilities.
- While MySQL offers `FULLTEXT` indexing on many types of table and column, those `FULLTEXT` indexes are *not* used to fulfill queries using `LIKE`.

Section 3.5: SELECT with CASE or IF

Query

```
SELECT st.name,
       st.percentage,
       CASE WHEN st.percentage >= 35 THEN 'Pass' ELSE 'Fail' END AS `Remark`
FROM student AS st ;
```

Result

```
+--------------------------------+
|   name   | percentage | Remark |
+--------------------------------+
|   Isha   |     67     |  Pass  |
|   Rucha  |     28     |  Fail  |
|   Het    |     35     |  Pass  |
|   Ansh   |     92     |  Pass  |
+--------------------------------+
```

Or with IF

```
SELECT st.name,
       st.percentage,
       IF(st.percentage >= 35, 'Pass', 'Fail') AS `Remark`
FROM student AS st ;
```

N.B

```
IF(st.percentage >= 35, 'Pass', 'Fail')
```

> This means : IF st.percentage >= 35 is **TRUE** then return `'Pass'` ELSE return `'Fail'`

Section 3.6: SELECT with Alias (AS)

SQL aliases are used to temporarily rename a table or a column. They are generally used to improve readability.

Query

```
SELECT username AS val FROM stack;
SELECT username val FROM stack;
```

(Note: `AS` is syntactically optional.)

Result

```
+-------+
| val   |
+-------+
| admin |
| stack |
+-------+
2 rows in set (0.00 sec)
```

Section 3.7: SELECT with a LIMIT clause

Query:

```
SELECT *
  FROM Customers
 ORDER BY CustomerID
 LIMIT 3;
```

Result:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Best Practice Always use `ORDER BY` when using `LIMIT`; otherwise the rows you will get will be unpredictable.

Query:

```
SELECT *
  FROM Customers
 ORDER BY CustomerID
 LIMIT 2,1;
```

Explanation:

When a `LIMIT` clause contains two numbers, it is interpreted as `LIMIT offset,count`. So, in this example the query skips two records and returns one.

Result:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Note:

The values in `LIMIT` clauses must be constants; they may not be column values.

Section 3.8: SELECT with BETWEEN

You can use BETWEEN clause to replace a combination of "greater than equal AND less than equal" conditions.

Data

```
+----+-----------+
| id | username  |
+----+-----------+
|  1 | admin     |
|  2 | root      |
|  3 | toor      |
|  4 | mysql     |
|  5 | thanks    |
|  6 | java      |
+----+-----------+
```

Query with operators

```
SELECT * FROM stack WHERE id >= 2 and id <= 5;
```

Similar query with BETWEEN

```
SELECT * FROM stack WHERE id BETWEEN 2 and 5;
```

Result

```
+----+-----------+
| id | username  |
+----+-----------+
|  2 | root      |
|  3 | toor      |
|  4 | mysql     |
|  5 | thanks    |
+----+-----------+
4 rows in set (0.00 sec)
```

Note

> **BETWEEN uses `>=` and `<=`, not `>` and `<`.**

Using NOT BETWEEN

If you want to use the negative you can use `NOT`. For example :

```
SELECT * FROM stack WHERE id NOT BETWEEN 2 and 5;
```

Result

```
+----+-----------+
| id | username  |
+----+-----------+
|  1 | admin     |
|  6 | java      |
+----+-----------+
2 rows in set (0.00 sec)
```

Note

> **NOT BETWEEN uses `>` and `<` and not `>=` and `<=`** That is, `WHERE id NOT BETWEEN 2 and 5` is the same as `WHERE (id < 2 OR id > 5)`.

If you have an index on a column you use in a `BETWEEN` search, MySQL can use that index for a range scan.

Section 3.9: SELECT with WHERE

Query

```
SELECT * FROM stack WHERE username = "admin" AND password = "admin";
```

Result

```
+------+----------+----------+
| id   | username | password |
+------+----------+----------+
|    1 | admin    | admin    |
+------+----------+----------+
1 row in set (0.00 sec)
```

Query with a nested SELECT in the WHERE clause

The `WHERE` clause can contain any valid `SELECT` statement to write more complex queries. This is a 'nested' query

Query

Nested queries are usually used to return single atomic values from queries for comparisons.

```
SELECT title FROM books WHERE author_id = (SELECT id FROM authors WHERE last_name = 'Bar' AND
first_name = 'Foo');
```

Selects all usernames with no email address

```
SELECT * FROM stack WHERE username IN (SELECT username FROM signups WHERE email IS NULL);
```

Disclaimer: Consider using joins for performance improvements when comparing a whole result set.

Section 3.10: SELECT with LIKE(_)

A _ character in a `LIKE` clause pattern matches a single character.

Query

```
SELECT username FROM users WHERE users LIKE 'admin_';
```

Result

```
+----------+
| username |
+----------+
| admin1   |
```

```
| admin2   |
| admin-   |
| adminA   |
+----------+
```

Section 3.11: SELECT with date range

```
SELECT ... WHERE dt >= '2017-02-01'
             AND dt  < '2017-02-01' + INTERVAL 1 MONTH
```

Sure, this could be done with `BETWEEN` and inclusion of `23:59:59`. But, the pattern has this benefits:

- You don't have pre-calculate the end date (which is often an exact length from the start)
- You don't include both endpoints (as `BETWEEN` does), nor type '23:59:59' to avoid it.
- It works for `DATE`, `TIMESTAMP`, `DATETIME`, and even the microsecond-included `DATETIME(6)`.
- It takes care of leap days, end of year, etc.
- It is index-friendly (so is `BETWEEN`).

Chapter 4: Backticks

Section 4.1: Backticks usage

There are many examples where backticks are used inside a query but for many it's still unclear when or where to use backticks ``.

Backticks are mainly used to prevent an error called "*MySQL reserved word*". When making a table in PHPmyAdmin you are sometimes faced with a warning or alert that you are using a "*MySQL reserved word*".

For example when you create a table with a column named "`group`" you get a warning. This is because you can make the following query:

```
SELECT student_name, AVG(test_score) FROM student GROUP BY group
```

To make sure you don't get an error in your query you have to use backticks so your query becomes:

```
SELECT student_name, AVG(test_score) FROM student GROUP BY `group`
```

Table

Not only column names can be surrounded by backticks, but also table names. For example when you need to `JOIN` multiple tables.

```
SELECT `users`.`username`, `groups`.`group` FROM `users`
```

Easier to read

As you can see using backticks around table and column names also make the query easier to read.

For example when you are used to write querys all in lower case:

```
select student_name, AVG(test_score) from student group by group
select `student_name`, AVG(`test_score`) from `student` group by `group`
```

Please see the MySQL Manual page entitled Keywords and Reserved Words. The ones with an (R) are Reserved Words. The others are merely Keywords. The Reserved require special caution.

Chapter 5: NULL

Section 5.1: Uses for NULL

- Data not yet known - such as `end_date`, `rating`
- Optional data - such as `middle_initial` (though that might be better as the empty string)
- 0/0 - The result of certain computations, such as zero divided by zero.
- NULL is not equal to "" (blank string) or 0 (in case of integer).
- others?

Section 5.2: Testing NULLs

- `IS NULL` / `IS NOT NULL` -- `= NULL` does not work like you expect.
- `x <=> y` is a "null-safe" comparison.

In a `LEFT JOIN` tests for rows of `a` for which there is *not* a corresponding row in `b`.

```
SELECT ...
    FROM a
    LEFT JOIN b ON ...
    WHERE b.id IS NULL
```

Chapter 6: Limit and Offset

Section 6.1: Limit and Offset relationship

Considering the following `users` table:

id	username
1	User1
2	User2
3	User3
4	User4
5	User5

In order to constrain the number of rows in the result set of a `SELECT` query, the `LIMIT` clause can be used together with one or two positive integers as arguments (zero included).

`LIMIT` clause with one argument

When one argument is used, the result set will only be constrained to the number specified in the following manner:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2
```

id	username
1	User1
2	User2

If the argument's value is `0`, the result set will be empty.

Also notice that the `ORDER BY` clause may be important in order to specify the first rows of the result set that will be presented (when ordering by another column).

`LIMIT`clause with two arguments

When two arguments are used in a `LIMIT` clause:

- the **first** argument represents the row from which the result set rows will be presented – this number is often mentioned as an **offset**, since it represents the row previous to the initial row of the constrained result set. This allows the argument to receive `0` as value and thus taking into consideration the first row of the non-constrained result set.
- the **second** argument specifies the maximum number of rows to be returned in the result set (similarly to the one argument's example).

Therefore the query:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2, 3
```

Presents the following result set:

id	username
3	User3
4	User4
5	User5

Notice that when the **offset** argument is `0`, the result set will be equivalent to a one argument `LIMIT` clause. This means that the following 2 queries:

```
SELECT * FROM users ORDER BY id ASC LIMIT 0, 2

SELECT * FROM users ORDER BY id ASC LIMIT 2
```

Produce the same result set:

id	username
1	User1
2	User2

`OFFSET` keyword: alternative syntax

An alternative syntax for the `LIMIT` clause with two arguments consists in the usage of the `OFFSET` keyword after the first argument in the following manner:

```
SELECT * FROM users ORDER BY id ASC LIMIT 2 OFFSET 3
```

This query would return the following result set:

id	username
3	User3
4	User4

Notice that in this alternative syntax the arguments have their positions switched:

- the **first** argument represents the number of rows to be returned in the result set;
- the **second** argument represents the offset.

Chapter 7: Creating databases

Parameter	Details
CREATE DATABASE	Creates a database with the given name
CREATE SCHEMA	This is a synonym for `CREATE DATABASE`
IF NOT EXISTS	Used to avoid execution error, if specified database already exists
create_specification	`create_specification` options specify database characteristics such as `CHARACTER SET` and `COLLATE`(database collation)

Section 7.1: Create database, users, and grants

Create a DATABASE. Note that the shortened word SCHEMA can be used as a synonym.

```
CREATE DATABASE Baseball; -- creates a database named Baseball
```

If the database already exists, Error 1007 is returned. To get around this error, try:

```
CREATE DATABASE IF NOT EXISTS Baseball;
```

Similarly,

```
DROP DATABASE IF EXISTS Baseball; -- Drops a database if it exists, avoids Error 1008
DROP DATABASE xyz; -- If xyz does not exist, ERROR 1008 will occur
```

Due to the above Error possibilities, DDL statements are often used with `IF EXISTS`.

One can create a database with a default CHARACTER SET and collation. For example:

```
CREATE DATABASE Baseball CHARACTER SET utf8 COLLATE utf8_general_ci;

SHOW CREATE DATABASE Baseball;
```

```
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| Baseball | CREATE DATABASE `Baseball` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+----------+-----------------------------------------------------------------+
```

See your current databases:

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| ajax_stuff         |
| Baseball           |
+--------------------+
```

Set the currently active database, and see some information:

```
USE Baseball; -- set it as the current database
```

```
SELECT @@character_set_database as cset,@@collation_database as col;
```

```
+------+-----------------+
| cset | col             |
+------+-----------------+
| utf8 | utf8_general_ci |
+------+-----------------+
```

The above shows the default CHARACTER SET and Collation for the database.

Create a user:

```
CREATE USER 'John123'@'%' IDENTIFIED BY 'OpenSesame';
```

The above creates a user John123, able to connect with any hostname due to the `%` wildcard. The Password for the user is set to 'OpenSesame' which is hashed.

And create another:

```
CREATE USER 'John456'@'%' IDENTIFIED BY 'somePassword';
```

Show that the users have been created by examining the special `mysql` database:

```
SELECT user,host,password from mysql.user where user in ('John123','John456');
```

```
+---------+------+-------------------------------------------+
| user    | host | password                                  |
+---------+------+-------------------------------------------+
| John123 | %    | *E6531C342ED87 .....................      |
| John456 | %    | *B04E11FAAAE9A .....................      |
+---------+------+-------------------------------------------+
```

Note that at this point, the users have been created, but without any permissions to use the Baseball database.

Work with permissions for users and databases. Grant rights to user John123 to have full privileges on the Baseball database, and just SELECT rights for the other user:

```
GRANT ALL ON Baseball.* TO 'John123'@'%';
GRANT SELECT ON Baseball.* TO 'John456'@'%';
```

Verify the above:

```
SHOW GRANTS FOR 'John123'@'%';
```

```
+-----------------------------------------------------------------------------------------------
-------+
| Grants for John123@%
|
+-----------------------------------------------------------------------------------------------
-------+
| GRANT USAGE ON *.* TO 'John123'@'%' IDENTIFIED BY PASSWORD '*E6531C342ED87 .....................
|
| GRANT ALL PRIVILEGES ON `baseball`.* TO 'John123'@'%'
|
+-----------------------------------------------------------------------------------------------
-------+
```

```
SHOW GRANTS FOR 'John456'@'%';
```

```
+-----------------------------------------------------------------------------------------------
-------+
| Grants for John456@%
|
+-----------------------------------------------------------------------------------------------
-------+
| GRANT USAGE ON *.* TO 'John456'@'%' IDENTIFIED BY PASSWORD '*B04E11FAAAE9A ..................
|
| GRANT SELECT ON `baseball`.* TO 'John456'@'%'
|
+-----------------------------------------------------------------------------------------------
-------+
```

Note that the **GRANT USAGE** that you will always see means simply that the user may login. That is all that that means.

Section 7.2: Creating and Selecting a Database

If the administrator creates your database for you when setting up your permissions, you can begin using it. Otherwise, you need to create it yourself:

```
mysql> CREATE DATABASE menagerie;
```

Under Unix, database names are case sensitive (unlike SQL keywords), so you must always refer to your database as menagerie, not as Menagerie, MENAGERIE, or some other variant. This is also true for table names. (Under Windows, this restriction does not apply, although you must refer to databases and tables using the same lettercase throughout a given query. However, for a variety of reasons, the recommended best practice is always to use the same lettercase that was used when the database was created.)

Creating a database does not select it for use; you must do that explicitly. To make menagerie the current database, use this statement:

```
mysql> USE menagerie
Database changed
```

Your database needs to be created only once, but you must select it for use each time you begin a mysql session. You can do this by issuing a USE statement as shown in the example. Alternatively, you can select the database on the command line when you invoke mysql. Just specify its name after any connection parameters that you might need to provide. For example:

```
shell> mysql -h host -u user -p menagerie
Enter password: ********
```

Section 7.3: MyDatabase

You *must* create your own database, and not use write to any of the existing databases. This is likely to be one of the very first things to do after getting connected the first time.

```
CREATE DATABASE my_db;
USE my_db;
CREATE TABLE some_table;
```

```
INSERT INTO some_table ...;
```

You can reference your table by qualifying with the database name: `my_db.some_table`.

Section 7.4: System Databases

The following databases exist for MySQL's use. You may read (**SELECT**) them, but you must not write (**INSERT**/**UPDATE**/**DELETE**) the tables in them. (There are a few exceptions.)

- `mysql` -- repository for **GRANT** info and some other things.
- `information_schema` -- The tables here are 'virtual' in the sense that they are actually manifested by in-memory structures. Their contents include the schema for all tables.
- `performance_schema` -- ?? [please accept, then edit]
- others?? (for MariaDB, Galera, TokuDB, etc)

Chapter 8: Using Variables

Section 8.1: Setting Variables

Here are some ways to set variables:

1. You can set a variable to a specific, string, number, date using SET

```
SET @var_string = 'my_var';
SET @var_num = '2'
SET @var_date = '2015-07-20';
```

2. you can set a variable to be the result of a select statement using :=

```
Select @var := '123';
```

(Note: You need to use := when assigning a variable not using the SET syntax, because in other statements, (select, update...) the "=" is used to compare, so when you add a colon before the "=", you are saying "This is not a comparison, this is a SET".)

3. You can set a variable to be the result of a select statement using INTO

(This was particularly helpful when I needed to dynamically choose which Partitions to query from)

```
SET @start_date = '2015-07-20';
SET @end_date = '2016-01-31';
```

```
#this gets the year month value to use as the partition names
SET @start_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @start_date));
SET @end_yearmonth = (SELECT EXTRACT(YEAR_MONTH FROM @end_date));

#put the partitions into a variable
SELECT GROUP_CONCAT(partition_name)
FROM information_schema.partitions p
WHERE table_name = 'partitioned_table'
AND SUBSTRING_INDEX(partition_name,'P',-1) BETWEEN @start_yearmonth AND @end_yearmonth
INTO @partitions;

#put the query in a variable. You need to do this, because mysql did not recognize my variable as a
variable in that position. You need to concat the value of the variable together with the rest of the
query and then execute it as a stmt.
SET @query =
CONCAT('CREATE TABLE part_of_partitioned_table (PRIMARY KEY(id))
SELECT partitioned_table.*
FROM partitioned_table PARTITION(', @partitions,')
JOIN users u USING(user_id)
WHERE date(partitioned_table.date) BETWEEN ', @start_date,' AND ', @end_date);

#prepare the statement from @query
PREPARE stmt FROM @query;
#drop table
DROP TABLE IF EXISTS tech.part_of_partitioned_table;
#create table using statement
EXECUTE stmt;
```

Section 8.2: Row Number and Group By using variables in Select Statement

Let's say we have a table `team_person` as below:

```
+======+===========+
| team |    person |
+======+===========+
|   A  |      John |
+------+-----------+
|   B  |     Smith |
+------+-----------+
|   A  |    Walter |
+------+-----------+
|   A  |     Louis |
+------+-----------+
|   C  | Elizabeth |
+------+-----------+
|   B  |     Wayne |
+------+-----------+
```

```
CREATE TABLE team_person AS SELECT 'A' team, 'John' person
UNION ALL  SELECT 'B' team,  'Smith' person
UNION ALL  SELECT 'A' team,  'Walter' person
UNION ALL  SELECT 'A' team,  'Louis' person
UNION ALL  SELECT 'C' team,  'Elizabeth' person
UNION ALL  SELECT 'B' team,  'Wayne' person;
```

To select the table `team_person` with additional `row_number` column, either

```
SELECT @row_no := @row_no+1 AS row_number, team, person
FROM team_person, (SELECT @row_no := 0) t;
```

OR

```
SET @row_no := 0;
SELECT  @row_no := @row_no + 1 AS row_number, team, person
FROM team_person;
```

will output the result below:

```
+============+======+===========+
| row_number | team |    person |
+============+======+===========+
|          1 |   A  |      John |
+------------+------+-----------+
|          2 |   B  |     Smith |
+------------+------+-----------+
|          3 |   A  |    Walter |
+------------+------+-----------+
|          4 |   A  |     Louis |
+------------+------+-----------+
|          5 |   C  | Elizabeth |
+------------+------+-----------+
|          6 |   B  |     Wayne |
+------------+------+-----------+
```

Finally, if we want to get the `row_number` group by column `team`

```
SELECT @row_no := IF(@prev_val = t.team, @row_no + 1, 1) AS row_number
   ,@prev_val := t.team AS team
   ,t.person
FROM team_person t,
  (SELECT @row_no := 0) x,
  (SELECT @prev_val := '') y
ORDER BY t.team ASC,t.person DESC;
```

```
+============+======+===========+
| row_number | team |    person |
+============+======+===========+
|          1 |   A  |    Walter |
+------------+------+-----------+
|          2 |   A  |     Louis |
+------------+------+-----------+
|          3 |   A  |      John |
+------------+------+-----------+
|          1 |   B  |     Wayne |
+------------+------+-----------+
|          2 |   B  |     Smith |
+------------+------+-----------+
|          1 |   C  | Elizabeth |
+------------+------+-----------+
```

Chapter 9: Comment MySQL

Section 9.1: Adding comments

There are three types of comment:

```
 # This comment continues to the end of line

-- This comment continues to the end of line

/* This is an in-line comment */

/*
This is a
multiple-line comment
*/
```

Example:

```
SELECT * FROM t1; -- this is comment

CREATE TABLE stack(
    /*id_user int,
    username varchar(30),
    password varchar(30)
    */
    id int
);
```

The `--` method requires that a space follows the `--` before the comment begins, otherwise it will be interpreted as a command and usually cause an error.

```
#This comment works
/*This comment works.*/
--This comment does not.
```

Section 9.2: Commenting table definitions

```
CREATE TABLE menagerie.bird (
    bird_id INT NOT NULL AUTO_INCREMENT,
    species VARCHAR(300) DEFAULT NULL COMMENT 'You can include genus, but never subspecies.',
    INDEX idx_species (species) COMMENT 'We must search on species often.',
    PRIMARY KEY (bird_id)
) ENGINE=InnoDB COMMENT 'This table was inaugurated on February 10th.';
```

Using an `=` after `COMMENT` is optional. (Official docs)

These comments, unlike the others, are saved with the schema and can be retrieved via `SHOW CREATE TABLE` or from `information_schema`.

Chapter 10: INSERT

Section 10.1: INSERT, ON DUPLICATE KEY UPDATE

```
INSERT INTO `table_name`
    (`index_field`, `other_field_1`, `other_field_2`)
    VALUES
    ('index_value', 'insert_value', 'other_value')
    ON DUPLICATE KEY UPDATE
        `other_field_1` = 'update_value',
        `other_field_2` = VALUES(`other_field_2`);
```

This will `INSERT` into `table_name` the specified values, but if the unique key already exists, it will update the `other_field_1` to have a new value.
Sometimes, when updating on duplicate key it comes in handy to use `VALUES()` in order to access the original value that was passed to the `INSERT` instead of setting the value directly. This way, you can set different values by using `INSERT` and `UPDATE`. See the example above where `other_field_1` is set to `insert_value` on `INSERT` or to `update_value` on `UPDATE` while `other_field_2` is always set to `other_value`.

Crucial for the Insert on Duplicate Key Update (IODKU) to work is the schema containing a unique key that will signal a duplicate clash. This unique key can be a Primary Key or not. It can be a unique key on a single column, or a multi-column (composite key).

Section 10.2: Inserting multiple rows

```
INSERT INTO `my_table` (`field_1`, `field_2`) VALUES
    ('data_1', 'data_2'),
    ('data_1', 'data_3'),
    ('data_4', 'data_5');
```

This is an easy way to add several rows at once with one `INSERT` statement.

This kind of 'batch' insert is much faster than inserting rows one by one. Typically, inserting 100 rows in a single batch insert this way is 10 times as fast as inserting them all individually.

Ignoring existing rows

When importing large datasets, it may be preferable under certain circumstances to skip rows that would usually cause the query to fail due to a column restraint e.g. duplicate primary keys. This can be done using `INSERT IGNORE`.

Consider following example database:

```
SELECT * FROM `people`;
--- Produces:
+----+------+
| id | name |
+----+------+
|  1 | john |
|  2 | anna |
+----+------+


INSERT IGNORE INTO `people` (`id`, `name`) VALUES
('2', 'anna'), --- Without the IGNORE keyword, this record would produce an error
('3', 'mike');
```

```
SELECT * FROM `people`;
--- Produces:
+----+--------+
| id |  name  |
+----+--------+
|  1 |  john  |
|  2 |  anna  |
|  3 |  mike  |
+----+--------+
```

The important thing to remember is that *INSERT IGNORE* will also silently skip other errors too, here is what Mysql official documentations says:

> Data conversions that would trigger errors abort the statement if IGNORE is not > specified. With IGNORE, invalid values are adjusted to the closest values and >inserted; warnings are produced but the statement does not abort.

Note: The section below is added for the sake of completeness, but is not considered best practice (this would fail, for example, if another column was added into the table).

If you specify the value of the corresponding column for all columns in the table, you can ignore the column list in the `INSERT` statement as follows:

```
INSERT INTO `my_table` VALUES
        ('data_1', 'data_2'),
        ('data_1', 'data_3'),
        ('data_4', 'data_5');
```

Section 10.3: Basic Insert

```
INSERT INTO `table_name` (`field_one`, `field_two`) VALUES ('value_one', 'value_two');
```

In this trivial example, `table_name` is where the data are to be added, `field_one` and `field_two` are fields to set data against, and `value_one` and `value_two` are the data to do against `field_one` and `field_two` respectively.

It's good practice to list the fields you are inserting data into within your code, as if the table changes and new columns are added, your insert would break should they not be there

Section 10.4: INSERT with AUTO_INCREMENT + LAST_INSERT_ID()

When a table has an `AUTO_INCREMENT PRIMARY KEY`, normally one does not insert into that column. Instead, specify all the other columns, then ask what the new id was.

```
CREATE TABLE t (
    id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
    this ...,
    that ...,
    PRIMARY KEY(id) );

INSERT INTO t (this, that) VALUES (..., ...);
SELECT LAST_INSERT_ID() INTO @id;
INSERT INTO another_table (..., t_id, ...) VALUES (..., @id, ...);
```

Note that `LAST_INSERT_ID()` is tied to the session, so even if multiple connections are inserting into the same table, each with get its own id.

Your client API probably has an alternative way of getting the `LAST_INSERT_ID()` without actually performing a `SELECT` and handing the value back to the client instead of leaving it in an `@variable` inside MySQL. Such is usually preferable.

Longer, more detailed, example

The "normal" usage of IODKU is to trigger "duplicate key" based on some `UNIQUE` key, not the `AUTO_INCREMENT PRIMARY KEY`. The following demonstrates such. Note that it does *not* supply the `id` in the INSERT.

Setup for examples to follow:

```
CREATE TABLE iodku (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(99) NOT NULL,
    misc INT NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
) ENGINE=InnoDB;

INSERT INTO iodku (name, misc)
    VALUES
    ('Leslie', 123),
    ('Sally', 456);
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  |  456 |
+----+--------+------+
```

The case of IODKU performing an "update" and `LAST_INSERT_ID()` retrieving the relevant `id`:

```
INSERT INTO iodku (name, misc)
    VALUES
    ('Sally', 3333)            -- should update
    ON DUPLICATE KEY UPDATE    -- `name` will trigger "duplicate key"
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc);
SELECT LAST_INSERT_ID();       -- picking up existing value
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                2 |
+------------------+
```

The case where IODKU performs an "insert" and `LAST_INSERT_ID()` retrieves the new `id`:

```
INSERT INTO iodku (name, misc)
    VALUES
    ('Dana', 789)          -- Should insert
```

```
    ON DUPLICATE KEY UPDATE
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc);
SELECT LAST_INSERT_ID();   -- picking up new value
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                3 |
+------------------+
```

Resulting table contents:

```
SELECT * FROM iodku;
```

```
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  | 3333 |  -- IODKU changed this
|  3 | Dana   |  789 |  -- IODKU added this
+----+--------+------+
```

Section 10.5: INSERT SELECT (Inserting data from another Table)

This is the basic way to insert data from another table with the SELECT statement.

```
INSERT INTO `tableA` (`field_one`, `field_two`)
   SELECT `tableB`.`field_one`, `tableB`.`field_two`
   FROM `tableB`
   WHERE `tableB`.clmn <> 'someValue'
   ORDER BY `tableB`.`sorting_clmn`;
```

You can `SELECT * FROM`, but then `tableA` and `tableB` *must* have matching column count and corresponding datatypes.

Columns with `AUTO_INCREMENT` are treated as in the `INSERT` with `VALUES` clause.

This syntax makes it easy to fill (temporary) tables with data from other tables, even more so when the data is to be filtered on the insert.

Section 10.6: Lost AUTO_INCREMENT ids

Several 'insert' functions can "burn" ids. Here is an example, using InnoDB (other Engines may work differently):

```
CREATE TABLE Burn (
    id SMALLINT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(99) NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
        ) ENGINE=InnoDB;

INSERT IGNORE INTO Burn (name) VALUES ('first'), ('second');
SELECT LAST_INSERT_ID();          -- 1
SELECT * FROM Burn ORDER BY id;
```

```
  +----+--------+
  |  1 | first  |
  |  2 | second |
  +----+--------+

INSERT IGNORE INTO Burn (name) VALUES ('second');  -- dup 'IGNOREd', but id=3 is burned
SELECT LAST_INSERT_ID();          -- Still "1" -- can't trust in this situation
SELECT * FROM Burn ORDER BY id;
  +----+--------+
  |  1 | first  |
  |  2 | second |
  +----+--------+

INSERT IGNORE INTO Burn (name) VALUES ('third');
SELECT LAST_INSERT_ID();           -- now "4"
SELECT * FROM Burn ORDER BY id;    -- note that id=3 was skipped over
  +----+--------+
  |  1 | first  |
  |  2 | second |
  |  4 | third  |    -- notice that id=3 has been 'burned'
  +----+--------+
```

Think of it (roughly) this way: First the insert looks to see how many rows *might* be inserted. Then grab that many values from the auto_increment for that table. Finally, insert the rows, using ids as needed, and burning any left overs.

The only time the leftover are recoverable is if the system is shutdown and restarted. On restart, effectively `MAX(id)` is performed. This may reuse ids that were burned or that were freed up by `DELETE`s of the highest id(s).

Essentially any flavor of **`INSERT`** (including **`REPLACE`**, which is **`DELETE`** + **`INSERT`**) can burn ids. In InnoDB, the global (not session!) variable `innodb_autoinc_lock_mode` can be used to control some of what is going on.

When "normalizing" long strings into an `AUTO INCREMENT id`, burning can easily happen. This *could* lead to overflowing the size of the **`INT`** you chose.

Chapter 11: DELETE

Parameter	Details
LOW_PRIORITY	If `LOW_PRIORITY` is provided, the delete will be delayed until there are no processes reading from the table
IGNORE	If `IGNORE` is provided, all errors encountered during the delete are ignored
table	The table from which you are going to delete records
WHERE conditions	The conditions that must be met for the records to be deleted. If no conditions are provided, then all records from the table will be deleted
ORDER BY expression	If `ORDER BY` is provided, records will be deleted in the given order
LIMIT	It controls the maximum number of records to delete from the table. Given `number_rows` will be deleted.

Section 11.1: Multi-Table Deletes

MySQL's `DELETE` statement can use the `JOIN` construct, allowing also to specify which tables to delete from. This is useful to avoid nested queries. Given the schema:

```
create table people
(    id int primary key,
    name varchar(100) not null,
    gender char(1) not null
);
insert people (id,name,gender) values
(1,'Kathy','f'),(2,'John','m'),(3,'Paul','m'),(4,'Kim','f');

create table pets
(    id int auto_increment primary key,
    ownerId int not null,
    name varchar(100) not null,
    color varchar(100) not null
);
insert pets(ownerId,name,color) values
(1,'Rover','beige'),(2,'Bubbles','purple'),(3,'Spot','black and white'),
(1,'Rover2','white');
```

id	name	gender
1	Kathy	f
2	John	m
3	Paul	m
4	Kim	f

id	ownerId	name	color
1	1	Rover	beige
2	2	Bubbles	purple
4	1	Rover2	white

If we want to remove Paul's pets, the statement

```
DELETE p2
FROM pets p2
WHERE p2.ownerId in (
    SELECT p1.id
    FROM people p1
    WHERE p1.name = 'Paul');
```

can be rewritten as:

```
DELETE p2    -- remove only rows from pets
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

1 row deleted
Spot is deleted from Pets

`p1` and `p2` are aliases for the table names, especially useful for long table names and ease of readability.

To remove both the person and the pet:

```
DELETE p1, p2     -- remove rows from both tables
FROM people p1
JOIN pets p2
ON p2.ownerId = p1.id
WHERE p1.name = 'Paul';
```

2 rows deleted
Spot is deleted from Pets
Paul is deleted from People

foreign keys

When the DELETE statement involes tables with a foreing key constrain the optimizer may process the tables in an order that does not follow the relationship. Adding for example a foreign key to the definition of `pets`

```
ALTER TABLE pets ADD CONSTRAINT `fk_pets_2_people` FOREIGN KEY (ownerId) references people(id) ON
DELETE CASCADE;
```

the engine may try to delete the entries from `people` before `pets`, thus causing the following error:

```
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails
(`test`.`pets`, CONSTRAINT `pets_ibfk_1` FOREIGN KEY (`ownerId`) REFERENCES `people` (`id`))
```

The solution in this case is to delete the row from `people` and rely on `InnoDB`'s `ON DELETE` capabilities to propagate the deletion:

```
DELETE FROM people
WHERE name = 'Paul';
```

2 rows deleted
Paul is deleted from People
Spot is deleted on cascade from Pets

Another solution is to temporarily disable the check on foreing keys:

```
SET foreign_key_checks = 0;
DELETE p1, p2 FROM people p1 JOIN pets p2 ON p2.ownerId = p1.id WHERE p1.name = 'Paul';
SET foreign_key_checks = 1;
```

Section 11.2: DELETE vs TRUNCATE

```
TRUNCATE tableName;
```

This will delete all the data and reset `AUTO_INCREMENT` index. It's much faster than `DELETE FROM tableName` on a huge dataset. It can be very useful during development/testing.

When you ***truncate*** a table SQL server doesn't delete the data, it drops the table and recreates it, thereby deallocating the pages so there is a chance to recover the truncated data before the pages where overwritten. (The space cannot immediately be recouped for `innodb_file_per_table=OFF`.)

Section 11.3: Multi-table DELETE

MySQL allows to specify from which table the matching rows must be deleted

```
-- remove only the employees
DELETE e
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'
```

```
-- remove employees and department
DELETE e, d
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'
```

```
-- remove from all tables (in this case same as previous)
DELETE
FROM Employees e JOIN Department d ON e.department_id = d.department_id
WHERE d.name = 'Sales'
```

Section 11.4: Basic delete

```
DELETE FROM `myTable` WHERE `someColumn` = 'something'
```

The `WHERE` clause is optional but without it all rows are deleted.

Section 11.5: Delete with Where clause

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one'
```

This will delete all rows from the table where the contents of the `field_one` for that row match 'value_one'

The `WHERE` clause works in the same way as a select, so things like `>`, `<`, `<>` or `LIKE` can be used.

Notice: It is necessary to use conditional clauses (WHERE, LIKE) in delete query. If you do not use any conditional clauses then all data from that table will be deleted.

Section 11.6: Delete all rows from a table

```
DELETE FROM table_name ;
```

This will delete everything, all rows from the table. It is the most basic example of the syntax. It also shows that `DELETE` statements should really be used with extra care as they may empty a table, if the `WHERE` clause is omitted.

Section 11.7: LIMITing deletes

```
DELETE FROM `table_name` WHERE `field_one` = 'value_one' LIMIT 1
```

This works in the same way as the 'Delete with Where clause' example, but it will stop the deletion once the limited number of rows have been removed.

If you are limiting rows for deletion like this, be aware that it will delete the first row which matches the criteria. It might not be the one you would expect, as the results can come back unsorted if they are not explicitly ordered.

Chapter 12: UPDATE

Section 12.1: Update with Join Pattern

Consider a production table called `questions_mysql` and a table `iwtQuestions` (imported worktable) representing the last batch of imported CSV data from a `LOAD DATA INFILE`. The worktable is truncated before the import, the data is imported, and that process is not shown here.

Update our production data using a join to our imported worktable data.

```
UPDATE questions_mysql q -- our real table for production
join iwtQuestions i -- imported worktable
ON i.qId = q.qId
SET q.closeVotes = i.closeVotes,
q.votes = i.votes,
q.answers = i.answers,
q.views = i.views;
```

Aliases `q` and `i` are used to abbreviate the table references. This eases development and readability.

`qId`, the Primary Key, represents the Stackoverflow question id. Four columns are updated for matching rows from the join.

Section 12.2: Basic Update

Updating one row

```
UPDATE customers SET email='luke_smith@email.com' WHERE id=1
```

This query updates the content of `email` in the `customers` table to the string `luke_smith@email.com` where the value of `id` is equal to 1. The old and new contents of the database table are illustrated below on the left and right respectively:

customers

id	firstname	lastname	email
1	Luke	Smith	luke@example.com
2	Anna	Carey	anna@example.com
3	Todd	Winters	todd@example.com

customers

id	firstname	lastname	email
1	Luke	Smith	luke_smith@email.com
2	Anna	Carey	anna@example.com
3	Todd	Winters	todd@example.com

Updating all rows

```
UPDATE customers SET lastname='smith'
```

This query update the content of `lastname` for every entry in the `customers` table. The old and new contents of the database table are illustrated below on the left and right respectively:

customers

id	firstname	lastname	email
1	Luke	Smith	luke@example.com
2	Anna	Carey	anna@example.com
3	Todd	Winters	todd@example.com

customers

id	firstname	lastname	email
1	Luke	Smith	luke@example.com
2	Anna	Smith	anna@example.com
3	Todd	Smith	todd@example.com

Notice: It is necessary to use conditional clauses (WHERE) in UPDATE query. If you do not use any conditional

clause then all records of that table's attribute will be updated. In above example new value (Smith) of lastname in customers table set to all rows.

Section 12.3: Bulk UPDATE

When updating multiple rows with different values it is much quicker to use a bulk update.

```
UPDATE people
SET name =
  (CASE id WHEN 1 THEN 'Karl'
           WHEN 2 THEN 'Tom'
           WHEN 3 THEN 'Mary'
   END)
WHERE id IN (1,2,3);
```

By bulk updating only one query can be sent to the server instead of one query for each row to update. The cases should contain all possible parameters looked up in the `WHERE` clause.

Section 12.4: UPDATE with ORDER BY and LIMIT

If the `ORDER BY` clause is specified in your update SQL statement, the rows are updated in the order that is specified.

If `LIMIT` clause is specified in your SQL statement, that places a limit on the number of rows that can be updated. There is no limit, if `LIMIT` clause not specified.

`ORDER BY` and `LIMIT` cannot be used for multi table update.

Syntax for the MySQL `UPDATE` with `ORDER BY` and `LIMIT` is,

```
UPDATE [ LOW_PRIORITY ] [ IGNORE ]
tableName
SET column1 = expression1,
    column2 = expression2,
    ...
[WHERE conditions]
[ORDER BY expression [ ASC | DESC ]]
[LIMIT row_count];

---> Example
UPDATE employees SET isConfirmed=1 ORDER BY joiningDate LIMIT 10
```

In the above example, 10 rows will be updated according to the order of employees `joiningDate`.

Section 12.5: Multiple Table UPDATE

In multiple table `UPDATE`, it updates rows in each specified tables that satisfy the conditions. Each matching row is updated once, even if it matches the conditions multiple times.

In multiple table `UPDATE`, `ORDER BY` and `LIMIT` cannot be used.

Syntax for multi table `UPDATE` is,

```
UPDATE [LOW_PRIORITY] [IGNORE]
table1, table2, ...
    SET column1 = expression1,
```

```
        column2 = expression2,
        ...
    [WHERE conditions]
```

For example consider two tables, `products` and `salesOrders`. In case, we decrease the quantity of a particular product from the sales order which is placed already. Then we also need to increase that quantity in our stock column of `products` table. This can be done in single SQL update statement like below.

```
UPDATE products, salesOrders
  SET salesOrders.Quantity = salesOrders.Quantity - 5,
      products.availableStock = products.availableStock + 5
WHERE products.productId = salesOrders.productId
  AND salesOrders.orderId = 100 AND salesOrders.productId = 20;
```

In the above example, quantity '5' will be reduced from the `salesOrders` table and the same will be increased in `products` table according to the **`WHERE`** conditions.

Chapter 13: ORDER BY

Section 13.1: Contexts

The clauses in a `SELECT` have a specific order:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...
    ORDER BY ...  -- goes here
    LIMIT ... OFFSET ...;

( SELECT ... ) UNION ( SELECT ... ) ORDER BY ...  -- for ordering the result of the UNION.

SELECT ... GROUP_CONCAT(DISTINCT x ORDER BY ... SEPARATOR ...) ...

ALTER TABLE ... ORDER BY ... -- probably useful only for MyISAM; not for InnoDB
```

Section 13.2: Basic

ORDER BY x

x can be any datatype.

- `NULL`s precede non-NULLs.
- The default is `ASC` (lowest to highest)
- Strings (`VARCHAR`, etc) are ordered according the `COLLATION` of the declaration
- `ENUM`s are ordered by the declaration order of its strings.

Section 13.3: ASCending / DESCending

```
ORDER BY x ASC  -- same as default
ORDER BY x DESC  -- highest to lowest
ORDER BY lastname, firstname  -- typical name sorting; using two columns
ORDER BY submit_date DESC  -- latest first
ORDER BY submit_date DESC, id ASC  -- latest first, but fully specifying order.
```

- `ASC` = ASCENDING, `DESC` = DESCENDING
- `NULL`s come first even for `DESC`.
- In the above examples, `INDEX(x)`, `INDEX(lastname, firstname)`, `INDEX(submit_date)` may significantly improve performance.

But... Mixing `ASC` and `DESC`, as in the last example, cannot use a composite index to benefit. Nor will `INDEX(submit_date DESC, id ASC)` help -- "`DESC`" is recognized syntactically in the `INDEX` declaration, but ignored.

Section 13.4: Some tricks

```
ORDER BY FIND_IN_SET(card_type, "MASTER-CARD,VISA,DISCOVER") -- sort 'MASTER-CARD' first.
ORDER BY x IS NULL, x  -- order by `x`, but put `NULLs` last.
```

Custom ordering

```
SELECT * FROM some_table WHERE id IN (118, 17, 113, 23, 72)
ORDER BY FIELD(id, 118, 17, 113, 23, 72);
```

Returns the result in the specified order of ids.

id	...
118	...
17	...
113	...
23	...
72	...

Useful if the ids are already sorted and you just need to retrieve the rows.

Chapter 14: Group By

Parameter	DETAILS
expression1, expression2, ... expression_n	The expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY clause.
aggregate_function	A function such as SUM, COUNT, MIN, MAX, or AVG functions.
tables	he tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.
WHERE conditions	Optional. The conditions that must be met for the records to be selected.

Section 14.1: GROUP BY using HAVING

```
SELECT department, COUNT(*) AS "Man_Power"
FROM employees
GROUP BY department
HAVING COUNT(*) >= 10;
```

Using `GROUP BY ... HAVING` to filter aggregate records is analogous to using `SELECT ... WHERE` to filter individual records.

You could also say `HAVING Man_Power >= 10` since `HAVING` understands "aliases".

Section 14.2: Group By using Group Concat

Group Concat is used in MySQL to get concatenated values of expressions with more than one result per column. Meaning, there are many rows to be selected back for one column such as `Name(1):Score(*)`

Name	Score
Adam	A+
Adam	A-
Adam	B
Adam	C+
Bill	D-
John	A-

```
SELECT Name, GROUP_CONCAT(Score ORDER BY Score desc SEPERATOR ' ') AS Grades
FROM   Grade
GROUP BY Name
```

Results:

```
+------+------------+
| Name | Grades     |
+------+------------+
| Adam | C+ B A- A+ |
| Bill | D-         |
| John | A-         |
+------+------------+
```

Section 14.3: Group By Using MIN function

Assume a table of employees in which each row is an employee who has a `name`, a `department`, and a `salary`.

```
SELECT department, MIN(salary) AS "Lowest salary"
FROM employees
GROUP BY department;
```

This would tell you which department contains the employee with the lowest salary, and what that salary is. Finding the `name` of the employee with the lowest salary in each department is a different problem, beyond the scope of this Example. See "groupwise max".

Section 14.4: GROUP BY with AGGREGATE functions

Table ORDERS

```
+---------+------------+----------+-------+--------+
| orderid | customerid | customer | total | items  |
+---------+------------+----------+-------+--------+
|       1 |          1 | Bob      |  1300 |     10 |
|       2 |          3 | Fred     |   500 |      2 |
|       3 |          5 | Tess     |  2500 |      8 |
|       4 |          1 | Bob      |   300 |      6 |
|       5 |          2 | Carly    |   800 |      3 |
|       6 |          2 | Carly    |  1000 |     12 |
|       7 |          3 | Fred     |   100 |      1 |
|       8 |          5 | Tess     | 11500 |     50 |
|       9 |          4 | Jenny    |   200 |      2 |
|      10 |          1 | Bob      |   500 |     15 |
+---------+------------+----------+-------+--------+
```

- **COUNT**

Return the **number of rows** that satisfy a specific criteria in `WHERE` clause.

E.g.: Number of orders for each customer.

```
SELECT customer, COUNT(*) as orders
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

```
+----------+--------+
| customer | orders |
+----------+--------+
| Bob      |      3 |
| Carly    |      2 |
| Fred     |      2 |
| Jenny    |      1 |
| Tess     |      2 |
+----------+--------+
```

- **SUM**

Return the **sum** of the selected column.

E.g.: Sum of the total and items for each customer.

```
SELECT customer, SUM(total) as sum_total, SUM(items) as sum_items
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

```
+----------+-----------+-----------+
| customer | sum_total | sum_items |
+----------+-----------+-----------+
| Bob      |      2100 |        31 |
| Carly    |      1800 |        15 |
| Fred     |       600 |         3 |
| Jenny    |       200 |         2 |
| Tess     |     14000 |        58 |
+----------+-----------+-----------+
```

- **AVG**

Return the **average** value of a column of numeric value.

E.g.: Average order value for each customers.

```
SELECT customer, AVG(total) as avg_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

```
+----------+-----------+
| customer | avg_total |
+----------+-----------+
| Bob      |       700 |
| Carly    |       900 |
| Fred     |       300 |
| Jenny    |       200 |
| Tess     |      7000 |
+----------+-----------+
```

- **MAX**

Return the **highest** value of a certain column or expression.

E.g.: Highest order total for each customers.

```
SELECT customer, MAX(total) as max_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

```
+----------+-----------+
| customer | max_total |
+----------+-----------+
```

```
| Bob      |      1300 |
| Carly    |      1000 |
| Fred     |       500 |
| Jenny    |       200 |
| Tess     |     11500 |
+----------+-----------+
```

- **MIN**

Return the **lowest** value of a certain column or expression.

E.g.: Lowest order total for each customers.

```
SELECT customer, MIN(total) as min_total
FROM orders
GROUP BY customer
ORDER BY customer
```

Result:

```
+----------+-----------+
| customer | min_total |
+----------+-----------+
| Bob      |       300 |
| Carly    |       800 |
| Fred     |       100 |
| Jenny    |       200 |
| Tess     |      2500 |
+----------+-----------+
```

Chapter 15: Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause ...

Recently, new versions of MySQL servers have begun to generate 1055 errors for queries that used to work. This topic explains those errors. The MySQL team has been working to retire the nonstandard extension to `GROUP BY`, or at least to make it harder for query writing developers to be burned by it.

Section 15.1: Misusing GROUP BY to return unpredictable results: Murphy's Law

```
SELECT item.item_id, uses.category,   /* nonstandard */
       COUNT(*) number_of_uses
  FROM item
  JOIN uses ON item.item_id, uses.item_id
 GROUP BY item.item_id
```

will show the rows in a table called item, and show the count of related rows in a table called uses. It will also show the value of a column called `uses.category`.

This query works in MySQL (before the `ONLY_FULL_GROUP_BY` flag appeared). It uses MySQL's nonstandard extension to `GROUP BY`.

But the query has a problem: if several rows in the `uses` table match the `ON` condition in the `JOIN` clause, MySQL returns the `category` column from just one of those rows. Which row? The writer of the query, and the user of the application, doesn't get to know that in advance. Formally speaking, it's *unpredictable*: MySQL can return any value it wants.

Unpredictable is like *random,* with one significant difference. One might expect a *random* choice to change from time to time. Therefore, if a choice were random, you might detect it during debugging or testing. The *unpredictable* result is worse: MySQL returns the same result each time you use the query, *until it doesn't.* Sometimes it's a new version of the MySQL server that causes a different result. Sometimes it's a growing table causing the problem. What can go wrong, will go wrong, and when you don't expect it. That's called Murphy's Law.

The MySQL team has been working to make it harder for developers to make this mistake. Newer versions of MySQL in the 5.7 sequence have a `sql_mode` flag called `ONLY_FULL_GROUP_BY`. When that flag is set, the MySQL server returns the 1055 error and refuses to run this kind of query.

Section 15.2: Misusing GROUP BY with SELECT *, and how to fix it

Sometimes a query looks like this, with a `*` in the `SELECT` clause.

```
SELECT item.*,     /* nonstandard */
       COUNT(*) number_of_uses
  FROM item
  JOIN uses ON item.item_id, uses.item_id
 GROUP BY item.item_id
```

Such a query needs to be refactored to comply with the `ONLY_FULL_GROUP_BY` standard.

To do this, we need a subquery that uses `GROUP BY` correctly to return the `number_of_uses` value for each `item_id`.

This subquery is short and sweet, because it only needs to look at the `uses` table.

```
                            SELECT item_id, COUNT(*) number_of_uses
                              FROM  uses
                             GROUP BY item_id
```

Then, we can join that subquery with the `item` table.

```
SELECT item.*, usecount.number_of_uses
  FROM item
  JOIN (
                            SELECT item_id, COUNT(*) number_of_uses
                              FROM  uses
                             GROUP BY item_id
       ) usecount ON item.item_id = usecount.item_id
```

This allows the `GROUP BY` clause to be simple and correct, and also allows us to use the `*` specifier.

Note: nevertheless, wise developers avoid using the `*` specifier in any case. It's usually better to list the columns you want in a query.

Section 15.3: ANY_VALUE()

```
SELECT item.item_id, ANY_VALUE(uses.tag) tag,
       COUNT(*) number_of_uses
 FROM item
 JOIN uses ON item.item_id, uses.item_id
GROUP BY item.item_id
```

shows the rows in a table called `item`, the count of related rows, and one of the values in the related table called `uses`.

You can think of this `ANY_VALUE()` function as a strange a kind of aggregate function. Instead of returning a count, sum, or maximum, it instructs the MySQL server to choose, arbitrarily, one value from the group in question. It's a way of working around Error 1055.

Be careful when using `ANY_VALUE()` in queries in production applications.

It really should be called `SURPRISE_ME()`. It returns the value of some row in the GROUP BY group. Which row it returns is indeterminate. That means it's entirely up to the MySQL server. Formally, it returns an unpredictable value.

The server doesn't choose a random value, it's worse than that. It returns the same value every time you run the query, until it doesn't. It can change, or not, when a table grows or shrinks, or when the server has more or less RAM, or when the server version changes, or when Mars is in retrograde (whatever that means), or for no reason at all.

You have been warned.

Section 15.4: Using and misusing GROUP BY

```
SELECT item.item_id, item.name,     /* not SQL-92 */
       COUNT(*) number_of_uses
 FROM item
 JOIN uses ON item.item_id, uses.item_id
```

```
 GROUP BY item.item_id
```

will show the rows in a table called `item`, and show the count of related rows in a table called `uses`. This works well, but unfortunately it's not standard SQL-92.

Why not? because the `SELECT` clause (and the `ORDER BY` clause) in `GROUP BY` queries must contain columns that are

1. mentioned in the `GROUP BY` clause, or
2. aggregate functions such as `COUNT()`, `MIN()`, and the like.

This example's `SELECT` clause mentions `item.name`, a column that does not meet either of those criteria. MySQL 5.6 and earlier will reject this query if the SQL mode contains `ONLY_FULL_GROUP_BY`.

This example query can be made to comply with the SQL-92 standard by changing the `GROUP BY` clause, like this.

```
SELECT item.item_id, item.name,
       COUNT(*) number_of_uses
  FROM item
  JOIN uses ON item.item_id, uses.item_id
 GROUP BY item.item_id, item.name
```

The later SQL-99 standard allows a `SELECT` statement to omit unaggregated columns from the group key if the DBMS can prove a functional dependence between them and the group key columns. Because `item.name` is functionally dependent on `item.item_id`, the initial example is valid SQL-99. MySQL gained a functional dependence prover in version 5.7. The original example works under `ONLY_FULL_GROUP_BY`.

Chapter 16: Joins

Section 16.1: Joins visualized

If you are a visually oriented person, this Venn diagram may help you understand the different types of `JOIN`s that exist within MySQL.

Section 16.2: JOIN with subquery ("Derived" table)

```
SELECT x, ...
    FROM ( SELECT y, ... FROM ... ) AS a
    JOIN tbl  ON tbl.x = a.y
    WHERE ...
```

This will evaluate the subquery into a temp table, then `JOIN` that to `tbl`.

Prior to 5.6, there could not be an index on the temp table. So, this was potentially very inefficient:

```
SELECT ...
    FROM ( SELECT y, ... FROM ... ) AS a
    JOIN ( SELECT x, ... FROM ... ) AS b  ON b.x = a.y
    WHERE ...
```

With 5.6, the optimizer figures out the best index and creates it on the fly. (This has some overhead, so it is still not 'perfect'.)

Another common paradigm is to have a subquery to initialize something:

```
SELECT
        @n := @n + 1,
        ...
    FROM ( SELECT @n := 0 ) AS initialize
    JOIN the_real_table
    ORDER BY ...
```

(Note: this is technically a `CROSS JOIN` (Cartesian product), as indicated by the lack of `ON`. However it is efficient because the subquery returns only one row that has to be matched to the n rows in `the_real_table`.)

Section 16.3: Full Outer Join

MySQL does not support the `FULL OUTER JOIN`, but there are ways to emulate one.

Setting up the data

```
-- ----------------------------
-- Table structure for `owners`
-- ----------------------------
DROP TABLE IF EXISTS `owners`;
CREATE TABLE `owners` (
`owner_id` int(11) NOT NULL AUTO_INCREMENT,
`owner` varchar(30) DEFAULT NULL,
PRIMARY KEY (`owner_id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;

-- ----------------------------
-- Records of owners
-- ----------------------------
INSERT INTO `owners` VALUES ('1', 'Ben');
INSERT INTO `owners` VALUES ('2', 'Jim');
INSERT INTO `owners` VALUES ('3', 'Harry');
INSERT INTO `owners` VALUES ('6', 'John');
INSERT INTO `owners` VALUES ('9', 'Ellie');
-- ----------------------------
-- Table structure for `tools`
-- ----------------------------
DROP TABLE IF EXISTS `tools`;
CREATE TABLE `tools` (
`tool_id` int(11) NOT NULL AUTO_INCREMENT,
`tool` varchar(30) DEFAULT NULL,
`owner_id` int(11) DEFAULT NULL,
PRIMARY KEY (`tool_id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=latin1;
-- ----------------------------
-- Records of tools
-- ----------------------------
INSERT INTO `tools` VALUES ('1', 'Hammer', '9');
INSERT INTO `tools` VALUES ('2', 'Pliers', '1');
INSERT INTO `tools` VALUES ('3', 'Knife', '1');
INSERT INTO `tools` VALUES ('4', 'Chisel', '2');
INSERT INTO `tools` VALUES ('5', 'Hacksaw', '1');
INSERT INTO `tools` VALUES ('6', 'Level', null);
INSERT INTO `tools` VALUES ('7', 'Wrench', null);
INSERT INTO `tools` VALUES ('8', 'Tape Measure', '9');
INSERT INTO `tools` VALUES ('9', 'Screwdriver', null);
INSERT INTO `tools` VALUES ('10', 'Clamp', null);
```

What do we want to see?

We want to get a list, in which we see who owns which tools, and which tools might not have an owner.

The queries

To accomplish this, we can combine two queries by using `UNION`. In this first query we are joining the tools on the owners by using a `LEFT JOIN`. This will add all of our owners to our resultset, doesn't matter if they actually own tools.

In the second query we are using a `RIGHT JOIN` to join the tools onto the owners. This way we manage to get all the tools in our resultset, if they are owned by no one their owner column will simply contain `NULL`. By adding a `WHERE`-clause which is filtering by `owners.owner_id IS NULL` we are defining the result as those datasets, which have not already been returned by the first query, as we are only looking for the data in the right joined table.

Since we are using `UNION ALL` the resultset of the second query will be attached to the first queries resultset.

```
SELECT `owners`.`owner`, tools.tool
FROM `owners`
LEFT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
UNION ALL
SELECT `owners`.`owner`, tools.tool
FROM `owners`
RIGHT JOIN `tools` ON `owners`.`owner_id` = `tools`.`owner_id`
WHERE `owners`.`owner_id` IS NULL;
```

```
+-------+--------------+
| owner | tool         |
+-------+--------------+
| Ben   | Pliers       |
| Ben   | Knife        |
| Ben   | Hacksaw      |
| Jim   | Chisel       |
| Harry | NULL         |
| John  | NULL         |
| Ellie | Hammer       |
| Ellie | Tape Measure |
| NULL  | Level        |
| NULL  | Wrench       |
| NULL  | Screwdriver  |
| NULL  | Clamp        |
+-------+--------------+
12 rows in set (0.00 sec)
```

Section 16.4: Retrieve customers with orders -- variations on a theme

This will get all the orders for all customers:

```
SELECT c.CustomerName, o.OrderID
    FROM Customers AS c
    INNER JOIN Orders AS o
        ON c.CustomerID = o.CustomerID
    ORDER BY c.CustomerName, o.OrderID;
```

This will count the number of orders for each customer:

```
SELECT c.CustomerName, COUNT(*) AS 'Order Count'
    FROM Customers AS c
    INNER JOIN Orders AS o
        ON c.CustomerID = o.CustomerID
    GROUP BY c.CustomerID;
    ORDER BY c.CustomerName;
```

Also, counts, but probably faster:

```
SELECT  c.CustomerName,
        ( SELECT COUNT(*) FROM Orders WHERE CustomerID = c.CustomerID ) AS 'Order Count'
    FROM Customers AS c
    ORDER BY c.CustomerName;
```

List only the customer with orders.

```
SELECT  c.CustomerName,
    FROM Customers AS c
    WHERE EXISTS ( SELECT * FROM Orders WHERE CustomerID = c.CustomerID )
    ORDER BY c.CustomerName;
```

Section 16.5: Joining Examples

Query to create table on db

```
CREATE TABLE `user` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(30) NOT NULL,
`course` smallint(5) unsigned DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;

CREATE TABLE `course` (
`id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
`name` varchar(50) NOT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Since we're using InnoDB tables and know that user.course and course.id are related, we can specify a foreign key relationship:

```
ALTER TABLE `user`
ADD CONSTRAINT `FK_course`
FOREIGN KEY (`course`) REFERENCES `course` (`id`)
ON UPDATE CASCADE;
```

Join Query (Inner Join)

```
SELECT user.name, course.name
FROM `user`
INNER JOIN `course` on user.course = course.id;
```

Chapter 17: JOINS: Join 3 table with the same name of id.

Section 17.1: Join 3 tables on a column with the same name

```
CREATE TABLE Table1 (
    id INT UNSIGNED NOT NULL,
    created_on DATE NOT NULL,
    PRIMARY KEY (id)
)
CREATE TABLE Table2 (
    id INT UNSIGNED NOT NULL,
    personName VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
)
CREATE TABLE Table3 (
    id INT UNSIGNED NOT NULL,
    accountName VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
)
```

after creating the tables you could do a select query to get the id's of all three tables that are the same

```
SELECT
    t1.id AS table1Id,
    t2.id AS table2Id,
    t3.id AS table3Id
FROM Table1 t1
LEFT JOIN Table2 t2 ON t2.id = t1.id
LEFT JOIN Table3 t3 ON t3.id = t1.id
```

Chapter 18: UNION

Section 18.1: Combining SELECT statements with UNION

You can combine the results of two identically structured queries with the `UNION` keyword.

For example, if you wanted a list of all contact info from two separate tables, `authors` and `editors`, for instance, you could use the `UNION` keyword like so:

```
select name, email, phone_number
from authors

union

select name, email, phone_number
from editors
```

Using `union` by itself will strip out duplicates. If you needed to keep duplicates in your query, you could use the `ALL` keyword like so: `UNION ALL`.

Section 18.2: Combining data with different columns

```
SELECT name, caption as title, year, pages FROM books
UNION
SELECT name, title, year, 0 as pages FROM movies
```

When combining 2 record sets with different columns then emulate the missing ones with default values.

Section 18.3: ORDER BY

If you need to sort the results of a UNION, use this pattern:

```
( SELECT ... )
UNION
( SELECT ... )
ORDER BY
```

Without the parentheses, the final ORDER BY would belong to the last SELECT.

Section 18.4: Pagination via OFFSET

When adding a LIMIT to a UNION, this is the pattern to use:

```
( SELECT ... ORDER BY x  LIMIT 10 )
UNION
( SELECT ... ORDER BY x  LIMIT 10 )
ORDER BY x  LIMIT 10
```

Since you cannot predict which SELECT(s) will the "10" will come from, you need to get 10 from each, then further whittle down the list, repeating both the `ORDER BY` and `LIMIT`.

For the 4th page of 10 items, this pattern is needed:

```
( SELECT ... ORDER BY x  LIMIT 40 )
```

```
UNION
( SELECT ... ORDER BY x  LIMIT 40 )
ORDER BY x  LIMIT 30, 10
```

That is, collect 4 page's worth in each **SELECT**, then do the **OFFSET** in the **UNION**.

Section 18.5: Combining and merging data on different MySQL tables with the same columns into unique rows and running query

This ***UNION ALL*** combines data from multiple tables and serve as a table name alias to use for your queries:

```
SELECT YEAR(date_time_column), MONTH(date_time_column), MIN(DATE(date_time_column)),
MAX(DATE(date_time_column)), COUNT(DISTINCT (ip)), COUNT(ip), (COUNT(ip) / COUNT(DISTINCT (ip))) AS
Ratio
FROM (
    (SELECT date_time_column, ip FROM server_log_1 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_2 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log_3 WHERE state = 'action' AND log_id = 150) UNION
ALL
    (SELECT date_time_column, ip FROM server_log WHERE state = 'action' AND log_id = 150)
) AS table_all
GROUP BY YEAR(date_time_column), MONTH(date_time_column);
```

Section 18.6: UNION ALL and UNION

SELECT 1,22,44 UNION SELECT 2,33,55

SELECT 1,22,44 UNION SELECT 2,33,55 UNION SELECT 2,33,55

The result is the same as above.

use UNION ALL

when

SELECT 1,22,44 UNION SELECT 2,33,55 UNION ALL SELECT 2,33,55

Chapter 19: Arithmetic

Section 19.1: Arithmetic Operators

MySQL provides the following arithmetic operators

Operator	Name	Example
`+`	Addition	`SELECT 3+5;` -> 8 `SELECT 3.5+2.5;` -> 6.0 `SELECT 3.5+2;` -> 5.5
`-`	Subtraction	`SELECT 3-5;` -> -2
`*`	Multiplication	`SELECT 3 * 5;` -> 15
`/`	Division	`SELECT 20 / 4;` -> 5 `SELECT 355 / 113;` -> 3.1416 `SELECT 10.0 / 0;` -> NULL
`DIV`	Integer Division	`SELECT 5 DIV 2;` -> 2
`%` or `MOD`	Modulo	`SELECT 7 % 3;` -> 1 `SELECT 15 MOD 4` -> 3 `SELECT 15 MOD -4` -> 3 `SELECT -15 MOD 4` -> -3 `SELECT -15 MOD -4` -> -3 `SELECT 3 MOD 2.5` -> 0.5

BIGINT

If the numbers in your arithmetic are all integers, MySQL uses the `BIGINT` (signed 64-bit) integer data type to do its work. For example:

`select (1024 * 1024 * 1024 * 1024 *1024 * 1024) + 1` -> 1,152,921,504,606,846,977

and

`select (1024 * 1024 * 1024 * 1024 *1024 * 1024 * 1024` -> `BIGINT` out of range error

DOUBLE

If any numbers in your arithmetic are fractional, MySQL uses 64-bit IEEE 754 floating point arithmetic. You must be careful when using floating point arithmetic, because many floating point numbers are, inherently, approximations rather than exact values.

Section 19.2: Mathematical Constants

Pi

The following returns the value of `PI` formatted to 6 decimal places. The actual value is good to `DOUBLE`;

```
SELECT PI();    -> 3.141593
```

Section 19.3: Trigonometry (SIN, COS)

Angles are in Radians, not Degrees. All computations are done in IEEE 754 64-bit floating point. All floating point computations are subject to small errors, known as machine ε (epsilon) errors, so avoid trying to compare them for equality. There is no way to avoid these errors when using floating point; they are built in to the technology.

If you use `DECIMAL` values in trigonometric computations, they are implicitly converted to floating point, and then

back to decimal.

Sine

Returns the sine of a number X expressed in radians

```
SELECT SIN(PI()); -> 1.2246063538224e-16
```

Cosine

Returns the cosine of X when X is given in radians

```
SELECT COS(PI()); -> -1
```

Tangent

Returns the tangent of a number X expressed in radians. Notice the result is very close to zero, but not exactly zero. This is an example of machine ε.

```
SELECT TAN(PI());   -> -1.2246063538224e-16
```

Arc Cosine (inverse cosine)

Returns the arc cosine of X if X is in the range `-1 to 1`

```
SELECT ACOS(1);    -> 0
SELECT ACOS(1.01); -> NULL
```

Arc Sine (inverse sine)

Returns the arc sine of X if X is in the range `-1 to 1`

```
SELECT ASIN(0.2); -> 0.20135792079033
```

Arc Tangent (inverse tangent)

`ATAN(x)` returns the arc tangent of a single number.

```
SELECT ATAN(2); -> 1.1071487177941
```

`ATAN2(X, Y)` returns the arc tangent of the two variables X and Y. It is similar to calculating the arc tangent of Y / X. But it is numerically more robust: t functions correctly when X is near zero, and the signs of both arguments are used to determine the quadrant of the result.

Best practice suggests writing formulas to use `ATAN2()` rather than `ATAN()` wherever possible.

```
ATAN2(1,1);    -> 0.7853981633974483 (45 degrees)
ATAN2(1,-1);   -> 2.356194490192345  (135 degrees)
ATAN2(0, -1);  -> PI  (180 degrees)  don't try ATAN(-1 / 0)... it won't work
```

Cotangent

Returns the cotangent of X

```
SELECT COT(12); -> -1.5726734063977
```

Conversion

```
SELECT RADIANS(90) -> 1.5707963267948966
SELECT SIN(RADIANS(90)) -> 1
SELECT DEGREES(1), DEGREES(PI()) -> 57.29577951308232, 180
```

Section 19.4: Rounding (ROUND, FLOOR, CEIL)

Round a decimal number to an integer value

For exact numeric values (e.g. `DECIMAL`): If the first decimal place of a number is 5 or higher, this function will round a number to the next integer *away from zero*. If that decimal place is 4 or lower, this function will round to the next integer value *closest to zero*.

```
SELECT ROUND(4.51) -> 5
SELECT ROUND(4.49) -> 4
SELECT ROUND(-4.51) -> -5
```

For approximate numeric values (e.g. `DOUBLE`): The result of the `ROUND()` function depends on the C library; on many systems, this means that `ROUND()` uses the *round to the nearest even* rule:

```
SELECT ROUND(45e-1) -> 4  -- The nearest even value is 4
SELECT ROUND(55e-1) -> 6  -- The nearest even value is 6
```

Round up a number

To round up a number use either the `CEIL()` or `CEILING()` function

```
SELECT CEIL(1.23)    -> 2
SELECT CEILING(4.83) -> 5
```

Round down a number

To round down a number, use the `FLOOR()` function

```
SELECT FLOOR(1.99) -> 1
```

FLOOR and CEIL go toward / away from -infinity:

```
SELECT FLOOR(-1.01), CEIL(-1.01) -> -2 and -1
SELECT FLOOR(-1.99), CEIL(-1.99) -> -2 and -1
```

Round a decimal number to a specified number of decimal places.

```
SELECT ROUND(1234.987, 2) -> 1234.99
SELECT ROUND(1234.987, -2) -> 1200
```

The discussion of up versus down and "5" applies, too.

Section 19.5: Raise a number to a power (POW)

To raise a number x to a power y, use either the `POW()` or `POWER()` functions

```
SELECT POW(2,2); => 4
SELECT POW(4,2); => 16
```

Section 19.6: Square Root (SQRT)

Use the `SQRT()` function. If the number is negative, `NULL` will be returned

```
SELECT SQRT(16); -> 4
SELECT SQRT(-3); -> NULL
```

Section 19.7: Random Numbers (RAND)

Generate a random number

To generate a pseudorandom floating point number between `0` and `1`, use the `RAND()` function

Suppose you have the following query

```
SELECT i, RAND() FROM t;
```

This will return something like this

i	RAND()
1	0.6191438870682
2	0.93845168309142
3	0.83482678498591

Random Number in a range

To generate a random number in the range a <= n <= b, you can use the following formula

```
FLOOR(a + RAND() * (b - a + 1))
```

For example, this will generate a random number between 7 and 12

```
SELECT FLOOR(7 + (RAND() * 6));
```

A simple way to randomly return the rows in a table:

```
SELECT * FROM tbl ORDER BY RAND();
```

These are **pseudorandom** numbers.

The pseudorandom number generator in MySQL is not cryptographically secure. That is, if you use MySQL to generate random numbers to be used as secrets, a determined adversary who knows you used MySQL will be able to guess your secrets more easily than you might believe.

Section 19.8: Absolute Value and Sign (ABS, SIGN)

Return the absolute value of a number

```
SELECT ABS(2);   -> 2
SELECT ABS(-46); -> 46
```

The `sign` of a number compares it to 0.

Sign	Result	Example
-1	n < 0	`SELECT SIGN(42);` -> 1
0	n = 0	`SELECT SIGN(0);` -> 0
1	n > 0	`SELECT SIGN(-3);` -> -1

```
SELECT SIGN(-423421); -> -1
```

Chapter 20: String operations

Name	Description
ASCII()	Return numeric value of left-most character
BIN()	Return a string containing binary representation of a number
BIT_LENGTH()	Return length of argument in bits
CHAR()	Return the character for each integer passed
CHAR_LENGTH()	Return number of characters in argument
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenate with separator
ELT()	Return string at index number
EXPORT_SET()	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
FIELD()	Return the index (position) of the first argument in the subsequent arguments
FIND_IN_SET()	Return the index position of the first argument within the second argument
FORMAT()	Return a number formatted to specified number of decimal places
FROM_BASE64()	Decode to a base-64 string and return result
HEX()	Return a hexadecimal representation of a decimal or string value
INSERT()	Insert a substring at the specified position up to the specified number of characters
INSTR()	Return the index of the first occurrence of substring
LCASE()	Synonym for LOWER()
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOAD_FILE()	Load the named file
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits set
MATCH	Perform full-text search
MID()	Return a substring starting from the specified position
NOT LIKE	Negation of simple pattern matching
NOT REGEXP	Negation of REGEXP
OCT()	Return a string containing octal representation of a number
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Pattern matching using regular expressions
REPEAT()	Repeat a string the specified number of times
REPLACE()	Replace occurrences of a specified string
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Synonym for REGEXP

RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
SOUNDEX()	Return a soundex string
SOUNDS LIKE	Compare sounds
SPACE()	Return a string of the specified number of spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string
TRIM()	Remove leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase
WEIGHT_STRING()	Return the weight string for a string

Section 20.1: LENGTH()

Return the length of the string in bytes. Since some characters may be encoded using more than one byte, if you want the length in characters see CHAR_LENGTH()

Syntax: LENGTH(str)

```
LENGTH('foobar') -- 6
LENGTH('fööbar') -- 8 -- contrast with CHAR_LENGTH(...) = 6
```

Section 20.2: CHAR_LENGTH()

Return the number of characters in the string

Syntax: CHAR_LENGTH(str)

```
CHAR_LENGTH('foobar') -- 6
CHAR_LENGTH('fööbar') -- 6 -- contrast with LENGTH(...) = 8
```

Section 20.3: HEX(str)

Convert the argument to hexadecimal. This is used for strings.

```
HEX('fööbar') -- 66F6F6626172 -- in "CHARACTER SET latin1" because "F6" is hex for ö
HEX('fööbar') -- 66C3B6C3B6626172  -- in "CHARACTER SET utf8 or utf8mb4" because "C3B6" is hex for ö
```

Section 20.4: SUBSTRING()

SUBSTRING (or equivalent: SUBSTR) returns the substring starting from the specified position and, optionally, with the specified length

Syntax: `SUBSTRING(str, start_position)`

```
SELECT SUBSTRING('foobarbaz', 4); -- 'barbaz'
```

```
SELECT SUBSTRING('foobarbaz' FROM 4); -- 'barbaz'

-- using negative indexing
SELECT SUBSTRING('foobarbaz', -6); -- 'barbaz'
SELECT SUBSTRING('foobarbaz' FROM -6); -- 'barbaz'
```

Syntax: `SUBSTRING(str, start_position, length)`

```
SELECT SUBSTRING('foobarbaz', 4, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz', FROM 4 FOR 3); -- 'bar'

-- using negative indexing
SELECT SUBSTRING('foobarbaz', -6, 3); -- 'bar'
SELECT SUBSTRING('foobarbaz' FROM -6 FOR 3); -- 'bar'
```

Section 20.5: UPPER() / UCASE()

Convert in uppercase the string argument

Syntax: UPPER(str)

```
UPPER('fOoBar') -- 'FOOBAR'
UCASE('fOoBar') -- 'FOOBAR'
```

Section 20.6: STR_TO_DATE - Convert string to date

With a column of one of the string types, named `my_date_field` with a value such as [the string] `07/25/2016`, the following statement demonstrates the use of the `STR_TO_DATE` function:

```
SELECT STR_TO_DATE(my_date_field, '%m/%d/%Y') FROM my_table;
```

You could use this function as part of `WHERE` clause as well.

Section 20.7: LOWER() / LCASE()

Convert in lowercase the string argument

Syntax: LOWER(str)

```
LOWER('fOoBar') -- 'foobar'
LCASE('fOoBar') -- 'foobar'
```

Section 20.8: REPLACE()

Convert in lowercase the string argument

Syntax: REPLACE(str, from_str, to_str)

```
REPLACE('foobarbaz', 'bar', 'BAR') -- 'fooBARbaz'
REPLACE('foobarbaz', 'zzz', 'ZZZ') -- 'foobarbaz'
```

Section 20.9: Find element in comma separated list

```
SELECT FIND_IN_SET('b','a,b,c');
```

Return value:

> 2

```
SELECT FIND_IN_SET('d','a,b,c');
```

Return value:

> 0

Chapter 21: Date and Time Operations

Section 21.1: Date arithmetic

```
NOW() + INTERVAL 1 DAY  -- This time tomorrow

CURDATE() - INTERVAL 4 DAY -- Midnight 4 mornings ago
```

Show the mysql questions stored that were asked 3 to 10 hours ago (180 to 600 minutes ago):

```
SELECT qId,askDate,minuteDiff
FROM
(   SELECT qId,askDate,
    TIMESTAMPDIFF(MINUTE,askDate,now()) as minuteDiff
    FROM questions_mysql
) xDerived
WHERE minuteDiff BETWEEN 180 AND 600
ORDER BY qId DESC
LIMIT 50;
```

```
+----------+---------------------+------------+
| qId      | askDate             | minuteDiff |
+----------+---------------------+------------+
| 38546828 | 2016-07-23 22:06:50 |        182 |
| 38546733 | 2016-07-23 21:53:26 |        195 |
| 38546707 | 2016-07-23 21:48:46 |        200 |
| 38546687 | 2016-07-23 21:45:26 |        203 |
| ...      |                     |            |
+----------+---------------------+------------+
```

MySQL manual pages for TIMESTAMPDIFF().

Beware Do not try to use expressions like `CURDATE() + 1` for date arithmetic in MySQL. They don't return what you expect, especially if you're accustomed to the Oracle database product. Use `CURDATE() + INTERVAL 1 DAY` instead.

Section 21.2: SYSDATE(), NOW(), CURDATE()

```
  SELECT SYSDATE();
```

This function returns the current date and time as a value in `'YYYY-MM-DD HH:MM:SS'` or `YYYYMMDDHHMMSS` format, depending on whether the function is used in a string or numeric context. It returns the date and time in the current time zone.

```
  SELECT NOW();
```

This function is a synonym for `SYSDATE()`.

```
  SELECT CURDATE();
```

This function returns the current date, without any time, as a value in `'YYYY-MM-DD'` or `YYYYMMDD` format, depending on whether the function is used in a string or numeric context. It returns the date in the current time zone.

Section 21.3: Testing against a date range

Although it is very tempting to use `BETWEEN` ... `AND` ... for a date range, it is problematical. Instead, this pattern avoids most problems:

```
WHERE x >= '2016-02-25'
  AND x  < '2016-02-25' + INTERVAL 5 DAY
```

Advantages:

- `BETWEEN` is 'inclusive' thereby including the final date or second.
- `23:59:59` is clumsy and wrong if you have microsecond resolution on a `DATETIME`.
- This pattern avoid dealing with leap years and other data calculations.
- It works whether `x` is `DATE`, `DATETIME` or `TIMESTAMP`.

Section 21.4: Extract Date from Given Date or DateTime Expression

```
SELECT DATE('2003-12-31 01:02:03');
```

The output will be:

```
2003-12-31
```

Section 21.5: Using an index for a date and time lookup

Many real-world database tables have many rows with `DATETIME` OR `TIMESTAMP` column values spanning a lot of time, including years or even decades. Often it's necessary to use a `WHERE` clause to retrieve some subset of that timespan. For example, we might want to retrieve rows for the date 1-September-2016 from a table.

An inefficient way to do that is this:

```
 WHERE DATE(x) = '2016-09-01'   /* slow! */
```

It's inefficient because it applies a function -- `DATE()` -- to the values of a column. That means MySQL must examine each value of x, and an index cannot be used.

A better way to do the operation is this

```
 WHERE x >= '2016-09-01'
   AND x <  '2016-09-01' + INTERVAL 1 DAY
```

This selects a range of values of `x` lying anywhere on the day in question, up until but *not including* (hence `<`) midnight on the next day.

If the table has an index on the x column, then the database server can perform a range scan on the index. That means it can quickly find the first relevant value of x, and then scan the index sequentially until it finds the last relevant value. An index range scan is much more efficient than the full table scan required by `DATE(x) = '2016-09-01`.

Don't be tempted to use this, even though it looks more efficient.

```
  WHERE x BETWEEN '2016-09-01' AND '2016-09-01' + INTERVAL 1 DAY  /*  wrong! */
```

It has the same efficiency as the range scan, but it will select rows with values of x falling exactly at midnight on 2-Sept-2016, which is not what you want.

Section 21.6: Now()

```
Select Now();
```

Shows the current server date and time.

```
Update `footable` set mydatefield = Now();
```

This will update the field `mydatefield` with current server date and time in server's configured timezone, e.g.

```
'2016-07-21 12:00:00'
```

Chapter 22: Handling Time Zones

Section 22.1: Retrieve the current date and time in a particular time zone

This fetches the value of `NOW()` in local time, in India Standard Time, and then again in UTC.

```
SELECT NOW();
SET time_zone='Asia/Kolkata';
SELECT NOW();
SET time_zone='UTC';
SELECT NOW();
```

Section 22.2: Convert a stored `DATE` or `DATETIME` value to another time zone

If you have a stored `DATE` or `DATETIME` (in a column somewhere) it was stored with respect to some time zone, but in MySQL the time zone is *not* stored with the value. So, if you want to convert it to another time zone, you can, but you must know the original time zone. Using `CONVERT_TZ()` does the conversion. This example shows rows sold in California in local time.

```
SELECT CONVERT_TZ(date_sold,'UTC','America/Los_Angeles') date_sold_local
  FROM sales
 WHERE state_sold = 'CA'
```

Section 22.3: Retrieve stored `TIMESTAMP` values in a particular time zone

This is really easy. All `TIMESTAMP` values are stored in universal time, and always converted to the present `time_zone` setting whenever they are rendered.

```
SET SESSION time_zone='America/Los_Angeles';
SELECT timestamp_sold
  FROM sales
 WHERE state_sold = 'CA'
```

Why is this? `TIMESTAMP` values are based on the venerable UNIX `time_t` data type. Those UNIX timestamps are stored as a number of seconds since `1970-01-01 00:00:00` UTC.

Notice `TIMESTAMP` values are stored in universal time. `DATE` and `DATETIME` values are stored in whatever local time was in effect when they were stored.

Section 22.4: What is my server's local time zone setting?

Each server has a default global time_zone setting, configured by the owner of the server machine. You can find out the current time zone setting this way:

```
SELECT @@time_zone
```

Unfortunately, that usually yields the value `SYSTEM`, meaning the MySQL time is governed by the server OS's time zone setting.

This sequence of queries (yes, it's a hack) gives you back the offset in minutes between the server's time zone

setting and UTC.

```
CREATE TEMPORARY TABLE times (dt DATETIME, ts TIMESTAMP);
SET time_zone = 'UTC';
INSERT INTO times VALUES(NOW(), NOW());
SET time_zone = 'SYSTEM';
SELECT dt, ts, TIMESTAMPDIFF(MINUTE, dt, ts)offset FROM times;
DROP TEMPORARY TABLE times;
```

How does this work? The two columns in the temporary table with different data types is the clue. `DATETIME` data types are always stored in local time in tables, and `TIMESTAMP`s in UTC. So the `INSERT` statement, performed when the time_zone is set to UTC, stores two identical date / time values.

Then, the SELECT statement, is done when the time_zone is set to server local time. `TIMESTAMP`s are always translated from their stored UTC form to local time in SELECT statements. `DATETIME`s are not. So the `TIMESTAMPDIFF(MINUTE...)` operation computes the difference between local and universal time.

Section 22.5: What time_zone values are available in my server?

To get a list of possible time_zone values in your MySQL server instance, use this command.

```
SELECT mysql.time_zone_name.name
```

Ordinarily, this shows the ZoneInfo list of time zones maintained by Paul Eggert at the Internet Assigned Numbers Authority. Worldwide there are appproximately 600 time zones.

Unix-like operating systems (Linux distributions, BSD distributions, and modern Mac OS distributions, for example) receive routine updates. Installing these updates on an operating system lets the MySQL instances running there track the changes in time zone and daylight / standard time changeovers.

If you get a much shorter list of time zone names, your server is either incompletely configured or running on Windows. Here are instructions for your server administrator to install and maintain the ZoneInfo list.

Chapter 23: Regular Expressions

A regular expression is a powerful way of specifying a pattern for a complex search.

Section 23.1: REGEXP / RLIKE

The `REGEXP` (or its synonym, `RLIKE`) operator allows pattern matching based on regular expressions.

Consider the following `employee` table:

```
+-------------+-------------+-------------+--------------+----------+
| EMPLOYEE_ID | FIRST_NAME  | LAST_NAME   | PHONE_NUMBER | SALARY   |
+-------------+-------------+-------------+--------------+----------+
|         100 | Steven      | King        | 515.123.4567 | 24000.00 |
|         101 | Neena       | Kochhar     | 515.123.4568 | 17000.00 |
|         102 | Lex         | De Haan     | 515.123.4569 | 17000.00 |
|         103 | Alexander   | Hunold      | 590.423.4567 |  9000.00 |
|         104 | Bruce       | Ernst       | 590.423.4568 |  6000.00 |
|         105 | David       | Austin      | 590.423.4569 |  4800.00 |
|         106 | Valli       | Pataballa   | 590.423.4560 |  4800.00 |
|         107 | Diana       | Lorentz     | 590.423.5567 |  4200.00 |
|         108 | Nancy       | Greenberg   | 515.124.4569 | 12000.00 |
|         109 | Daniel      | Faviet      | 515.124.4169 |  9000.00 |
|         110 | John        | Chen        | 515.124.4269 |  8200.00 |
+-------------+-------------+-------------+--------------+----------+
```

Pattern ^

Select all employees whose `FIRST_NAME` starts with **N**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^N'
-- Pattern start with--------------------------^
```

Pattern $**

Select all employees whose `PHONE_NUMBER` ends with **4569**.

Query

```
SELECT * FROM employees WHERE PHONE_NUMBER REGEXP '4569$'
-- Pattern end with---------------------------------^
```

NOT REGEXP

Select all employees whose `FIRST_NAME` *does not* start with **N**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME NOT REGEXP '^N'
-- Pattern does not start with--------------^
```

Regex Contain

Select all employees whose `LAST_NAME` contains **in** and whose `FIRST_NAME` contains `a`.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP 'a' AND LAST_NAME REGEXP 'in'
-- No ^ or $, pattern can be anywhere ------------------------------------^
```

Any character between [ ]

Select all employees whose `FIRST_NAME` starts with **A** or **B** or **C**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]'
------------------------------------------------^^---^
```

Pattern or |

Select all employees whose `FIRST_NAME` starts with **A** or **B** or **C** and ends with **r**, **e**, or **i**.

Query

```
SELECT * FROM employees WHERE FIRST_NAME REGEXP '^[ABC]|[rei]$'
-- ------------------------------------------------^^---^^^---^^
```

Counting regular expression matches

Consider the following query:

```
SELECT FIRST_NAME, FIRST_NAME REGEXP '^N' as matching FROM employees
```

`FIRST_NAME REGEXP '^N'` is *1* or *0* depending on the fact that `FIRST_NAME` matches `^N`.

To visualize it better:

```
SELECT
FIRST_NAME,
IF(FIRST_NAME REGEXP '^N', 'matches ^N', 'does not match ^N') as matching
FROM employees
```

Finally, count total number of matching and non-matching rows with:

```
SELECT
IF(FIRST_NAME REGEXP '^N', 'matches ^N', 'does not match ^N') as matching,
COUNT(*)
FROM employees
GROUP BY matching
```

Chapter 24: VIEW

Parameters	Details
view_name	Name of View
SELECT statement	SQL statements to be packed in the views. It can be a SELECT statement to fetch data from one or more tables.

Section 24.1: Create a View

Privileges

The CREATE VIEW statement requires the CREATE VIEW privilege for the view, and some privilege for each column selected by the SELECT statement. For columns used elsewhere in the SELECT statement, you must have the SELECT privilege. If the OR REPLACE clause is present, you must also have the DROP privilege for the view. CREATE VIEW might also require the SUPER privilege, depending on the DEFINER value, as described later in this section.

When a view is referenced, privilege checking occurs.

A view belongs to a database. By default, a new view is created in the default database. To create the view explicitly in a given database, use a fully qualified name

For Example:

db_name.view_name

```
mysql> CREATE VIEW test.v AS SELECT * FROM t;
```

Note - Within a database, base tables and views share the same namespace, so a base table and a view cannot have the same name.

A VIEW can:

- be created from many kinds of SELECT statements
- refer to base tables or other views
- use joins, UNION, and subqueries
- SELECT need not even refer to any tables

Another Example

The following example defines a view that selects two columns from another table as well as an expression calculated from those columns:

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
```

```
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

Restrictions

- Before MySQL 5.7.7, the SELECT statement cannot contain a subquery in the FROM clause.
- The SELECT statement cannot refer to system variables or user-defined variables.
- Within a stored program, the SELECT statement cannot refer to program parameters or local variables.
- The SELECT statement cannot refer to prepared statement parameters.
- Any table or view referred to in the definition must exist. After the view has been created, it is possible to drop a table or view that
  the definition refers to. In this case, use of the view results in an error. To check a view definition for problems of this kind, use the CHECK TABLE statement.
- The definition cannot refer to a TEMPORARY table, and you cannot
  create a TEMPORARY view.
- You cannot associate a trigger with a view.
- Aliases for column names in the SELECT statement are checked against the maximum column length of 64 characters (not the maximum alias
  length of 256 characters).
- A **VIEW** may or may not optimize as well as the equivalent **SELECT**. It is unlikely to optimize any better.

Section 24.2: A view from two tables

A view is most useful when it can be used to pull in data from more than one table.

```
CREATE VIEW myview AS
SELECT a.*, b.extra_data FROM main_table a
LEFT OUTER JOIN other_table b
ON a.id = b.id
```

In mysql views are not materialized. If you now perform the simple query `SELECT * FROM myview`, mysql will actually perform the LEFT JOIN behind the scene.

A view once created can be joined to other views or tables

Section 24.3: DROPPING A VIEW

-- Create and drop a view in the current database.

```
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;
```

-- Create and drop a view referencing a table in a different database.

```
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT NULL;
DROP VIEW table_from_other_db;
```

Section 24.4: Updating a table via a VIEW

A **VIEW** acts very much like a table. Although you can **UPDATE** a table, you may or may not be able to update a view into that table. In general, if the **SELECT** in the view is complex enough to require a temp table, then **UPDATE** is not allowed.

Things like **GROUP BY**, **UNION**, **HAVING**, **DISTINCT**, and some subqueries prevent the view from being updatable. Details in *reference manual*.

Chapter 25: Table Creation

Section 25.1: Table creation with Primary Key

```
CREATE TABLE Person (
    PersonID     INT UNSIGNED NOT NULL,
    LastName     VARCHAR(66) NOT NULL,
    FirstName    VARCHAR(66),
    Address      VARCHAR(255),
    City         VARCHAR(66),
    PRIMARY KEY (PersonID)
);
```

A **primary key** is a `NOT NULL` single or a multi-column identifier which uniquely identifies a row of a table. An index is created, and if not explicitly declared as `NOT NULL`, MySQL will declare them so silently and implicitly.

A table can have only one `PRIMARY KEY`, and each table is recommended to have one. InnoDB will automatically create one in its absence, (as seen in MySQL documentation) though this is less desirable.

Often, an `AUTO_INCREMENT INT` also known as "surrogate key", is used for thin index optimization and relations with other tables. This value will (normally) increase by 1 whenever a new record is added, starting from a default value of 1.

However, despite its name, it is not its purpose to guarantee that values are incremental, merely that they are sequential and unique.

An auto-increment `INT` value will not reset to its default start value if all rows in the table are deleted, unless the table is truncated using `TRUNCATE TABLE` statement.

Defining one column as Primary Key (inline definition)

If the primary key consists of a single column, the `PRIMARY KEY` clause can be placed inline with the column definition:

```
CREATE TABLE Person (
    PersonID     INT UNSIGNED NOT NULL PRIMARY KEY,
    LastName     VARCHAR(66) NOT NULL,
    FirstName    VARCHAR(66),
    Address      VARCHAR(255),
    City         VARCHAR(66)
);
```

This form of the command is shorter and easier to read.

Defining a multiple-column Primary Key

It is also possible to define a primary key comprising more than one column. This might be done e.g. on the child table of a foreign-key relationship. A multi-column primary key is defined by listing the participating columns in a separate `PRIMARY KEY` clause. Inline syntax is not permitted here, as only one column may be declared `PRIMARY KEY` inline. For example:

```
CREATE TABLE invoice_line_items (
    LineNum      SMALLINT UNSIGNED NOT NULL,
    InvoiceNum   INT UNSIGNED NOT NULL,
    -- Other columns go here
```

```
    PRIMARY KEY (InvoiceNum, LineNum),
    FOREIGN KEY (InvoiceNum) REFERENCES -- references to an attribute of a table
);
```

Note that the columns of the primary key *should* be specified in logical sort order, which *may* be different from the order in which the columns were defined, as in the example above.

Larger indexes require more disk space, memory, and I/O. Therefore keys should be as small as possible (especially regarding composed keys). In InnoDB, every 'secondary index' includes a copy of the columns of the `PRIMARY KEY`.

Section 25.2: Basic table creation

The `CREATE TABLE` statement is used to create a table in a MySQL database.

```
CREATE TABLE Person (
    `PersonID`      INTEGER NOT NULL PRIMARY KEY,
    `LastName`      VARCHAR(80),
    `FirstName`     VARCHAR(80),
    `Address`       TEXT,
    `City`          VARCHAR(100)
) Engine=InnoDB;
```

Every field definition must have:

1. Field name: A valid field Name. Make sure to encolse the names in `-chars. This ensures that you can use eg space-chars in the fieldname.
2. Data type [Length]: If the field is `CHAR` or `VARCHAR`, it is mandatory to specify a field length.
3. Attributes `NULL` | `NOT NULL`: If `NOT NULL` is specified, then any attempt to store a `NULL` value in that field will fail.
4. See more on data types and their attributes here.

`Engine=...` is an optional parameter used to specify the table's storage engine. If no storage engine is specified, the table will be created using the server's default table storage engine (usually InnoDB or MyISAM).

Setting defaults

Additionally, where it makes sense you can set a default value for each field by using `DEFAULT`:

```
CREATE TABLE Address (
    `AddressID`    INTEGER NOT NULL PRIMARY KEY,
    `Street`       VARCHAR(80),
    `City`         VARCHAR(80),
    `Country`      VARCHAR(80) DEFAULT "United States",
    `Active`       BOOLEAN DEFAULT 1,
) Engine=InnoDB;
```

If during inserts no `Street` is specified, that field will be `NULL` when retrieved. When no `Country` is specified upon insert, it will default to "United States".

You can set default values for all column types, except for `BLOB`, `TEXT`, `GEOMETRY`, and `JSON` fields.

Section 25.3: Table creation with Foreign Key

```
CREATE TABLE Account (
    AccountID      INT UNSIGNED NOT NULL,
    AccountNo      INT UNSIGNED NOT NULL,
```

```
    PersonID    INT UNSIGNED,
    PRIMARY KEY (AccountID),
    FOREIGN KEY (PersonID) REFERENCES Person (PersonID)
) ENGINE=InnoDB;
```

Foreign key: A Foreign Key (FK) is either a single column, or multi-column composite of columns, in a *referencing* table. This FK is confirmed to exist in the *referenced* table. It is highly recommended that the *referenced* table key confirming the FK be a Primary Key, but that is not enforced. It is used as a fast-lookup into the *referenced* where it does not need to be unique, and in fact can be a left-most index there.

Foreign key relationships involve a parent table that holds the central data values, and a child table with identical values pointing back to its parent. The FOREIGN KEY clause is specified in the child table. The parent and child tables must use the same storage engine. They must not be TEMPORARY tables.

Corresponding columns in the foreign key and the referenced key must have similar data types. The size and sign of integer types must be the same. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.

Note: foreign-key constraints are supported under the InnoDB storage engine (not MyISAM or MEMORY). DB set-ups using other engines will accept this `CREATE TABLE` statement but will not respect foreign-key constraints. (Although newer MySQL versions default to `InnoDB`, but it is good practice to be explicit.)

Section 25.4: Show Table Structure

If you want to see the schema information of your table, you can use one of the following:

```
SHOW CREATE TABLE child; -- Option 1

CREATE TABLE `child` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `fullName` varchar(100) NOT NULL,
  `myParent` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `mommy_daddy` (`myParent`),
  CONSTRAINT `mommy_daddy` FOREIGN KEY (`myParent`) REFERENCES `parent` (`id`)
      ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

If used from the mysql commandline tool, this is less verbose:

```
SHOW CREATE TABLE child \G
```

A less descriptive way of showing the table structure:

```
mysql> CREATE TABLE Tab1(id int, name varchar(30));
Query OK, 0 rows affected (0.03 sec)

mysql> DESCRIBE Tab1; -- Option 2

+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | YES  |     | NULL    |       |
| name  | varchar(30) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

> Both **DESCRIBE** and **DESC** gives the same result.

To see `DESCRIBE` performed on all tables in a database at once, see this Example.

Section 25.5: Cloning an existing table

A table can be replicated as follows:

```
CREATE TABLE ClonedPersons LIKE Persons;
```

The new table will have exactly the same structure as the original table, including indexes and column attributes.

As well as manually creating a table, it is also possible to create table by selecting data from another table:

```
CREATE TABLE ClonedPersons SELECT * FROM Persons;
```

You can use any of the normal features of a `SELECT` statement to modify the data as you go:

```
CREATE TABLE ModifiedPersons
SELECT PersonID, FirstName + LastName AS FullName FROM Persons
WHERE LastName IS NOT NULL;
```

Primary keys and indexes will not be preserved when creating tables from `SELECT`. You must redeclare them:

```
CREATE TABLE ModifiedPersons (PRIMARY KEY (PersonID))
SELECT PersonID, FirstName + LastName AS FullName FROM Persons
WHERE LastName IS NOT NULL;
```

Section 25.6: Table Create With TimeStamp Column To Show Last Update

The TIMESTAMP column will show when the row was last updated.

```
CREATE TABLE `TestLastUpdate` (
    `ID` INT NULL,
    `Name` VARCHAR(50) NULL,
    `Address` VARCHAR(50) NULL,
    `LastUpdate` TIMESTAMP NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
COMMENT='Last Update'
;
```

Section 25.7: CREATE TABLE FROM SELECT

You can create one table from another by adding a `SELECT` statement at the end of the `CREATE TABLE` statement:

```
CREATE TABLE stack (
    id_user INT,
    username VARCHAR(30),
    password VARCHAR(30)
);
```

Create a table in the same database:

```sql
-- create a table from another table in the same database with all attributes
CREATE TABLE stack2 AS SELECT * FROM stack;

-- create a table from another table in the same database with some attributes
CREATE TABLE stack3 AS SELECT username, password FROM stack;
```

Create tables from different databases:

```sql
-- create a table from another table from another database with all attributes
CREATE TABLE stack2 AS SELECT * FROM second_db.stack;

-- create a table from another table from another database with some attributes
CREATE TABLE stack3 AS SELECT username, password FROM second_db.stack;
```

N.B

To create a table same of another table that exist in another database, you need to specifies the name of the database like this:

```sql
FROM NAME_DATABASE.name_table
```

Chapter 26: ALTER TABLE

Section 26.1: Changing storage engine; rebuild table; change file_per_table

For example, if `t1` is currently not an InnoDB table, this statement changes its storage engine to InnoDB:

```
ALTER TABLE t1 ENGINE = InnoDB;
```

If the table is already InnoDB, this will rebuild the table and its indexes and have an effect similar to `OPTIMIZE TABLE`. You may gain some disk space improvement.

If the value of `innodb_file_per_table` is currently different than the value in effect when `t1` was built, this will convert to (or from) file_per_table.

Section 26.2: ALTER COLUMN OF TABLE

```
CREATE DATABASE stackoverflow;

USE stackoverflow;

Create table stack(
    id_user int NOT NULL,
    username varchar(30) NOT NULL,
    password varchar(30) NOT NULL
);

ALTER TABLE stack ADD COLUMN submit date NOT NULL; -- add new column
ALTER TABLE stack DROP COLUMN submit; -- drop column
ALTER TABLE stack MODIFY submit DATETIME NOT NULL; -- modify type column
ALTER TABLE stack CHANGE submit submit_date DATETIME NOT NULL; -- change type and name of column
ALTER TABLE stack ADD COLUMN mod_id INT NOT NULL AFTER id_user; -- add new column after existing
column
```

Section 26.3: Change auto-increment value

Changing an auto-increment value is useful when you don't want a gap in an AUTO_INCREMENT column after a massive deletion.

For example, you got a lot of unwanted (advertisement) rows posted in your table, you deleted them, and you want to fix the gap in auto-increment values. Assume the MAX value of AUTO_INCREMENT column is 100 now. You can use the following to fix the auto-increment value.

```
ALTER TABLE your_table_name AUTO_INCREMENT = 101;
```

Section 26.4: Renaming a MySQL table

Renaming a table can be done in a single command:

```
RENAME TABLE `<old name>` TO `<new name>`;
```

The following syntax does exactly the same:

```
ALTER TABLE `<old name>` RENAME TO `<new name>`;
```

If renaming a temporary table, the `ALTER TABLE` version of the syntax must be used.

Steps:

1. Replace **<old** name**>** and **<new** name**>** in the line above with the relevant values. *Note: If the table is being moved to a different database, the dbname.tablename syntax can be used for* ***<old*** *name***>** *and/or* ***<new*** *name***>****.*
2. Execute it on the relevant database in the MySQL command line or a client such as MySQL Workbench. *Note: The user must have ALTER and DROP privileges on the old table and CREATE and INSERT on the new one.*

Section 26.5: ALTER table add INDEX

To improve performance one might want to add indexes to columns

```
ALTER TABLE TABLE_NAME ADD INDEX `index_name` (`column_name`)
```

altering to add composite (multiple column) indexes

```
ALTER TABLE TABLE_NAME ADD INDEX `index_name` (`col1`,`col2`)
```

Section 26.6: Changing the type of a primary key column

```
ALTER TABLE fish_data.fish DROP PRIMARY KEY;
ALTER TABLE fish_data.fish MODIFY COLUMN fish_id DECIMAL(20,0) NOT NULL PRIMARY KEY;
```

An attempt to modify the type of this column without first dropping the primary key would result in an error.

Section 26.7: Change column definition

The change the definition of a db column, the query below can be used for example, if we have this db schema

```
users (
    firstname varchar(20),
    lastname varchar(20),
    age char(2)
)
```

To change the type of `age` column from `char` to `int`, we use the query below:

```
ALTER TABLE users CHANGE age age tinyint UNSIGNED NOT NULL;
```

General format is:

```
ALTER TABLE table_name CHANGE column_name new_column_definition
```

Section 26.8: Renaming a MySQL database

There is no single command to rename a MySQL database but a simple workaround can be used to achieve this by backing up and restoring:

```
mysqladmin -uroot -p<password> create <new name>
mysqldump -uroot -p<password> --routines <old name> | mysql -uroot -pmypassword <new name>
mysqladmin -uroot -p<password> drop <old name>
```

Steps:

1. Copy the lines above into a text editor.
2. Replace all references to **<old** name**>**, **<new** name**>** and **<password>** (+ optionally `root` to use a different user) with the relevant values.
3. Execute one by one on the command line (assuming the MySQL "bin" folder is in the path and entering "y" when prompted).

Alternative Steps:

Rename (move) each table from one db to the other. Do this for each table:

```
RENAME TABLE `<old db>`.`<name>` TO  `<new db>`.`<name>`;
```

You can create those statements by doing something like

```
SELECT CONCAT('RENAME TABLE old_db.', table_name, ' TO ',
                          'new_db.', table_name)
    FROM information_schema.TABLES
    WHERE table_schema = 'old_db';
```

Warning. Do not attempt to do any sort of table or database by simply moving files around on the filesystem. This worked fine in the old days of just MyISAM, but in the new days of InnoDB and tablespaces, it won't work. Especially when the "Data Dictionary" is moved from the filesystem into system InnoDB tables, probably in the next major release. Moving (as opposed to just `DROPping`) a **PARTITION** of an InnoDB table requires using "transportable tablespaces". In the near future, there won't even be a file to reach for.

Section 26.9: Swapping the names of two MySQL databases

The following commands can be used to swap the names of two MySQL databases (**<db1>** and **<db2>**):

```
mysqladmin -uroot -p<password> create swaptemp
mysqldump -uroot -p<password> --routines <db1> | mysql -uroot -p<password> swaptemp
mysqladmin -uroot -p<password> drop <db1>
mysqladmin -uroot -p<password> create <db1>
mysqldump -uroot -p<password> --routines <db2> | mysql -uroot -p<password> <db1>
mysqladmin -uroot -p<password> drop <db2>
mysqladmin -uroot -p<password> create <db2>
mysqldump -uroot -p<password> --routines swaptemp | mysql -uroot -p<password> <db2>
mysqladmin -uroot -p<password> drop swaptemp
```

Steps:

1. Copy the lines above into a text editor.
2. Replace all references to **<db1>**, **<db2>** and **<password>** (+ optionally `root` to use a different user) with the relevant values.
3. Execute one by one on the command line (assuming the MySQL "bin" folder is in the path and entering "y" when prompted).

Section 26.10: Renaming a column in a MySQL table

Renaming a column can be done in a single statement but as well as the new name, the "column definition" (i.e. its data type and other optional properties such as nullability, auto incrementing etc.) must also be specified.

```
ALTER TABLE `<table name>` CHANGE `<old name>` `<new name>` <column definition>;
```

Steps:

1. Open the MySQL command line or a client such as MySQL Workbench.
2. Run the following statement: `SHOW CREATE TABLE <table name>;` (replacing **`<table name>`** with the relevant value).
3. Make a note of the entire column definition for the column to be renamed *(i.e. everything that appears after the name of the column but before the comma separating it from the next column name)*.
4. Replace **`<old name>`**, **`<new name>`** and **`<column definition>`** in the line above with the relevant values and then execute it.

Chapter 27: Drop Table

Parameters	Details
TEMPORARY	Optional. It specifies that only temporary tables should be dropped by the DROP TABLE statement.
IF EXISTS	Optional. If specified, the DROP TABLE statement will not raise an error if one of the tables does not exist.

Section 27.1: Drop Table

Drop Table is used to delete the table from database.

Creating Table:

Creating a table named tbl and then deleting the created table

```
CREATE TABLE tbl(
    id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR(100) NOT NULL,
    author VARCHAR(40) NOT NULL,
    submission_date DATE,
    PRIMARY KEY (id)
);
```

Dropping Table:

```
DROP TABLE tbl;
```

> **PLEASE NOTE**
>
> Dropping table will completely delete the table from the database and all its information, and it will not be recovered.

Section 27.2: Drop tables from database

```
DROP TABLE Database.table_name
```

Chapter 28: MySQL LOCK TABLE

Section 28.1: Row Level Locking

If the tables use InnoDB, MySQL automatically uses row level locking so that multiple transactions can use same table simultaneously for read and write, without making each other wait.

If two transactions trying to modify the same row and both uses row level locking, one of the transactions waits for the other to complete.

Row level locking also can be obtained by using `SELECT ... FOR UPDATE` statement for each rows expected to be modified.

Consider two connections to explain Row level locking in detail

Connection 1

```
START TRANSACTION;
SELECT ledgerAmount FROM accDetails WHERE id = 1 FOR UPDATE;
```

In connection 1, row level lock obtained by `SELECT ... FOR UPDATE` statement.

Connection 2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1;
```

When some one try to update same row in connection 2, that will wait for connection 1 to finish transaction or error message will be displayed according to the `innodb_lock_wait_timeout` setting, which defaults to 50 seconds.

```
Error Code: 1205. Lock wait timeout exceeded; try restarting transaction
```

To view details about this lock, run `SHOW ENGINE INNODB STATUS`

```
---TRANSACTION 1973004, ACTIVE 7 sec updating
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 360, 1 row lock(s)
MySQL thread id 4, OS thread handle 0x7f996beac700, query id 30 localhost root update
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1
------- TRX HAS BEEN WAITING 7 SEC FOR THIS LOCK TO BE GRANTED:
```

Connection 2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 250 WHERE id=2;
```

```
1 row(s) affected
```

But while updating some other row in connection 2 will be executed without any error.

Connection 1

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 750 WHERE id=1;
COMMIT;
```

```
1 row(s) affected
```

Now row lock is released, because transaction is commited in Connection 1.

Connection 2

```
UPDATE accDetails SET ledgerAmount = ledgerAmount + 500 WHERE id=1;
```

```
1 row(s) affected
```

The update is executed without any error in Connection 2 after Connection 1 released row lock by finishing the transaction.

Section 28.2: Mysql Locks

Table locks can be an important tool for ***ENGINE=****MyISAM, but are rarely useful for* ***ENGINE=InnoDB****. If you are tempted to use table locks with InnoDB, you should rethink how you are working with transactions.*

MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables, or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them. A session can acquire or release locks only for itself. One session cannot acquire locks for another session or release locks held by another session.

Locks may be used to emulate transactions or to get more speed when updating tables. This is explained in more detail later in this section.

Command:`LOCK TABLES table_name READ|WRITE;`

you can assign only lock type to a single table;

Example (READ LOCK):

```
LOCK TABLES table_name READ;
```

Example (WRITE LOCK):

```
LOCK TABLES table_name WRITE;
```

To see lock is applied or not, use following Command

```
SHOW OPEN TABLES;
```

To flush/remove all locks, use following command:

```
UNLOCK TABLES;
```

EXAMPLE:

```
LOCK TABLES products WRITE:
INSERT INTO products(id,product_name) SELECT id,old_product_name FROM old_products;
UNLOCK TABLES;
```

Above example any external connection cannot write any data to products table until unlocking table product

EXAMPLE:

```
LOCK TABLES products READ:
INSERT INTO products(id,product_name) SELECT id,old_product_name FROM old_products;
UNLOCK TABLES;
```

Above example any external connection cannot read any data from products table until unlocking table product

Chapter 29: Error codes

Section 29.1: Error code 1064: Syntax error

```
select LastName, FirstName,
from Person
```

Returns message:

> Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'from Person' at line 2.

Getting a "1064 error" message from MySQL means the query cannot be parsed without syntax errors. In other words it can't make sense of the query.

The quotation in the error message begins with the first character of the query that MySQL can't figure out how to parse. In this example MySQL can't make sense, in context, of `from Person`. In this case, there's an extra comma immediately before `from Person`. The comma tells MySQL to expect another column description in the `SELECT` clause

A syntax error always says `... near '...'`. The thing at the beginning of the quotes is very near where the error is. To locate an error, look at the first token in the quotes and at the last token before the quotes.

Sometimes you will get `... near ''`; that is, nothing in the quotes. That means the first character MySQL can't figure out is right at the end or the beginning of the statement. This suggests the query contains unbalanced quotes (`'` or `"`) or unbalanced parentheses or that you did not terminate the statement before correctly.

In the case of a Stored Routine, you may have forgotten to properly use `DELIMITER`.

So, when you get Error 1064, look at the text of the query, and find the point mentioned in the error message. Visually inspect the text of the query right around that point.

If you ask somebody to help you troubleshoot Error 1064, it's best to provide both the text of the whole query and the text of the error message.

Section 29.2: Error code 1175: Safe Update

This error appears while trying to update or delete records without including the `WHERE` clause that uses the `KEY` column.

To execute the delete or update anyway - type:

```
SET SQL_SAFE_UPDATES = 0;
```

To enable the safe mode again - type:

```
SET SQL_SAFE_UPDATES = 1;
```

Section 29.3: Error code 1215: Cannot add foreign key

constraint

This error occurs when tables are not adequately structured to handle the speedy lookup verification of Foreign Key (FK) requirements that the developer is mandating.

```
CREATE TABLE `gtType` (
  `type` char(2) NOT NULL,
  `description` varchar(1000) NOT NULL,
  PRIMARY KEY (`type`)
) ENGINE=InnoDB;

CREATE TABLE `getTogethers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `type` char(2) NOT NULL,
  `eventDT` datetime NOT NULL,
  `location` varchar(1000) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_gt2type` (`type`), -- see Note1 below
  CONSTRAINT `gettogethers_ibfk_1` FOREIGN KEY (`type`) REFERENCES `gtType` (`type`)
) ENGINE=InnoDB;
```

Note1: a KEY like this will be created automatically if needed due to the FK definition in the line that follows it. The developer can skip it, and the KEY (a.k.a. index) will be added if necessary. An example of it being skipped by the developer is shown below in `someOther`.

So far so good, until the below call.

```
CREATE TABLE `someOther` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `someDT` datetime NOT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `someOther_dt` FOREIGN KEY (`someDT`) REFERENCES `getTogethers` (`eventDT`)
) ENGINE=InnoDB;
```

> Error Code: 1215. Cannot add foreign key constraint

In this case it fails due to the lack of an index in the *referenced* table `getTogethers` to handle the speedy lookup of an `eventDT`. To be solved in next statement.

```
CREATE INDEX `gt_eventdt` ON getTogethers (`eventDT`);
```

Table `getTogethers` has been modified, and now the creation of `someOther` will succeed.

From the MySQL Manual Page Using FOREIGN KEY Constraints:

> MySQL requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. In the referencing table, there must be an index where the foreign key columns are listed as the first columns in the same order. Such an index is created on the referencing table automatically if it does not exist.
>
> Corresponding columns in the foreign key and the referenced key must have similar data types. The size and sign of integer types must be the same. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.

> InnoDB permits a foreign key to reference any index column or group of columns. However, in the referenced table, there must be an index where the referenced columns are listed as the first columns in the same order.

Note that last point above about first (left-most) columns and the lack of a Primary Key requirement (though highly advised).

Upon successful creation of a *referencing* (child) table, any keys that were automatically created for you are visible with a command such as the following:

```
SHOW CREATE TABLE someOther;
```

Other common cases of experiencing this error include, as mentioned above from the docs, but should be highlighted:

- Seemingly trivial differences in `INT` which is signed, pointing toward `INT UNSIGNED`.
- Developers having trouble understanding multi-column (composite) KEYS and first (left-most) ordering requirements.

Section 29.4: 1067, 1292, 1366, 1411 - Bad Value for number, date, default, etc

1067 This is probably related to `TIMESTAMP` defaults, which have changed over time. See `TIMESTAMP defaults` in the Dates & Times page. (which does not exist yet)

1292/1366 DOUBLE/Integer Check for letters or other syntax errors. Check that the columns align; perhaps you think you are putting into a `VARCHAR` but it is aligned with a numeric column.

1292 DATETIME Check for too far in past or future. Check for between 2am and 3am on a morning when Daylight savings changed. Check for bad syntax, such as `+00` timezone stuff.

1292 VARIABLE Check the allowed values for the `VARIABLE` you are trying to `SET`.

1292 LOAD DATA Look at the line that is 'bad'. Check the escape symbols, etc. Look at the datatypes.

1411 STR_TO_DATE Incorrectly formatted date?

Section 29.5: 1045 Access denied

See discussions in "GRANT" and "Recovering root password".

Section 29.6: 1236 "impossible position" in Replication

Usually this means that the Master crashed and that `sync_binlog` was OFF. The solution is to `CHANGE MASTER to POS=0` of the next binlog file (see the Master) on the Slave.

The cause: The Master sends replication items to the Slave before flushing to its binlog (when `sync_binlog=OFF`). If the Master crashes before the flush, the Slave has already logically moved past the end of file on the binlog. When the Master starts up again, it starts a new binlog, so CHANGEing to the beginning of that binlog is the best available solution.

A longer term solution is `sync_binlog=ON`, if you can afford the extra I/O that it causes.

(If you are running with GTID, ...?)

Section 29.7: 2002, 2003 Cannot connect

Check for a Firewall issue blocking port 3306.

Some possible diagnostics and/or solutions

- Is the server actually running?
- "service firewalld stop" and "systemctl disable firewalld"
- telnet master 3306
- Check the `bind-address`
- check `skip-name-resolve`
- check the socket.

Section 29.8: 126, 127, 134, 144, 145

When you try access the records from MySQL database, you may get these error messages. These error messages occurred due to corruption in MySQL database. Following are the types

```
MySQL error code 126 = Index file is crashed
MySQL error code 127 = Record-file is crashed
MySQL error code 134 = Record was already deleted (or record file crashed)
MySQL error code 144 = Table is crashed and last repair failed
MySQL error code 145 = Table was marked as crashed and should be repaired
```

MySQL bug, virus attack, server crash, improper shutdown, damaged table are the reason behind this corruption. When it gets corrupted, it becomes inaccessible and you cannot access them anymore. In order to get accessibility, the best way to retrieve data from an updated backup. However, if you do not have updated or any valid backup then you can go for MySQL Repair.

If the table engine type is `MyISAM`, apply **`CHECK TABLE`**, then **`REPAIR TABLE`** to it.

Then think seriously about converting to InnoDB, so this error won't happen again.

Syntax

```
CHECK TABLE <table name> ////To check the extent of database corruption
REPAIR TABLE <table name> ////To repair table
```

Section 29.9: 139

Error 139 may mean that the number and size of the fields in the table definition exceeds some limit. Workarounds:

- Re-think the schema
- Normalize some fields
- Vertically partition the table

Section 29.10: 1366

This usually means that the character set handling was not consistent between client and server. See ... for further assistance.

Section 29.11: 126, 1054, 1146, 1062, 24

(taking a break) With the inclusion of those 4 error numbers, I think this page will have covered about 50% of the typical errors users get.

(Yes, this 'Example' needs revision.)

24 Can't open file (Too many open files)

`open_files_limit` comes from an OS setting. `table_open_cache` needs to be less than that.

These can cause that error:

- Failure to `DEALLOCATE PREPARE` in a stored procedure.
- PARTITIONed table(s) with a large number of partitions and innodb_file_per_table = ON. Recommend not having more than 50 partitions in a given table (for various reasons). (When "Native Partitions" become available, this advice may change.)

The obvious workaround is to set increase the OS limit: To allow more files, change `ulimit` or `/etc/security/limits.conf` or in `sysctl.conf` (kern.maxfiles & kern.maxfilesperproc) or something else (OS dependent). Then increase `open_files_limit` and `table_open_cache`.

As of 5.6.8, `open_files_limit` is auto-sized based on `max_connections`, but it is OK to change it from the default.

1062 - Duplicate Entry

This error occur mainly because of the following two reasons

1. *Duplicate Value* - `Error Code: 1062. Duplicate entry '12' for key 'PRIMARY'`

   The primary key column is unique and it will not accept the duplicate entry. So when you are trying to insert a new row which is already present in you table will produce this error.

> To solve this, Set the primary key column as `AUTO_INCREMENT`. And when you are trying to insert a new row, ignore the primary key column or insert `NULL` value to primary key.

```
CREATE TABLE userDetails(
  userId INT(10) NOT NULL AUTO_INCREMENT,
  firstName VARCHAR(50),
  lastName VARCHAR(50),
  isActive INT(1) DEFAULT 0,
  PRIMARY KEY (userId) );

--->and now while inserting
INSERT INTO userDetails VALUES (NULL ,'John', 'Doe', 1);
```

2. *Unique data field* - `Error Code: 1062. Duplicate entry 'A' for key 'code'`

   You may assigned a column as unique and trying to insert a new row with already existing value for that column will produce this error.

> To overcome this error, use **INSERT IGNORE** instead of normal **INSERT**. If the new row which you are trying to insert doesn't duplicate an existing record, MySQL inserts it as usual. If the record is a duplicate, the **IGNORE** keyword discard it without generating any error.

```
INSERT IGNORE INTO userDetails VALUES (NULL ,'John', 'Doe', 1);
```

Chapter 30: Stored routines (procedures and functions)

Parameter	Details
RETURNS	Specifies the data type that can be returned from a function.
RETURN	Actual variable or value following the `RETURN` syntax is what is returned to where the function was called from.

Section 30.1: Stored procedure with IN, OUT, INOUT parameters

```
DELIMITER $$

DROP PROCEDURE IF EXISTS sp_nested_loop$$
CREATE PROCEDURE sp_nested_loop(IN i INT, IN j INT, OUT x INT, OUT y INT, INOUT z INT)
BEGIN
    DECLARE a INTEGER DEFAULT 0;
    DECLARE b INTEGER DEFAULT 0;
    DECLARE c INTEGER DEFAULT 0;
    WHILE a < i DO
        WHILE b < j DO
            SET c = c + 1;
            SET b = b + 1;
        END WHILE;
        SET a = a + 1;
        SET b = 0;
    END WHILE;
    SET x = a, y = c;
    SET z = x + y + z;
END $$
DELIMITER ;
```

Invokes (CALL) the stored procedure:

```
SET @z = 30;
call sp_nested_loop(10, 20, @x, @y, @z);
SELECT @x, @y, @z;
```

Result:

```
+------+------+------+
|  @x  |  @y  |  @z  |
+------+------+------+
|  10  |  200 |  240 |
+------+------+------+
```

An `IN` parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.

An **`OUT`** parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.

An **`INOUT`** parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

Ref: http://dev.mysql.com/doc/refman/5.7/en/create-procedure.html

Section 30.2: Create a Function

The following (trivial) example function simply returns the constant `INT` value `12`.

```
DELIMITER ||
CREATE FUNCTION functionname()
RETURNS INT
BEGIN
    RETURN 12;
END;
||
DELIMITER ;
```

The first line defines what the delimiter character(`DELIMITER ||`) is to be changed to, this is needed to be set before a function is created otherwise if left it at its default `;` then the first `;` that is found in the function body will be taken as the end of the `CREATE` statement, which is usually not what is desired.

After the `CREATE FUNCTION` has run you should set the delimiter back to its default of `;` as is seen after the function code in the above example (`DELIMITER ;`).

Execution this function is as follows:

```
SELECT functionname();
```

```
+----------------+
| functionname() |
+----------------+
|             12 |
+----------------+
```

A slightly more complex (but still trivial) example takes a parameter and adds a constant to it:

```
DELIMITER $$
CREATE FUNCTION add_2 ( my_arg INT )
  RETURNS INT
BEGIN
  RETURN (my_arg + 2);
END;
$$
DELIMITER ;

SELECT add_2(12);
+-----------+
| add_2(12) |
+-----------+
|        14 |
+-----------+
```

Note the use of a different argument to the `DELIMITER` directive. You can actually use any character sequence that does not appear in the `CREATE` statement body, but the usual practice is to use a doubled non-alphanumeric character such as `\\`, `||` or `$$`.

It is good practice to always change the parameter before and after a function, procedure or trigger creation or update as some GUI's don't require the delimiter to change whereas running queries via the command line always require the delimiter to be set.

Section 30.3: Cursors

Cursors enable you to itterate results of query one by line. `DECLARE` command is used to init cursor and associate it with a specific SQL query:

```
DECLARE student CURSOR FOR SELECT name FROM studend;
```

Let's say we sell products of some types. We want to count how many products of each type are exists.

Our data:

```
CREATE TABLE product
(
  id   INT(10) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  type VARCHAR(50)      NOT NULL,
  name VARCHAR(255)     NOT NULL

);
CREATE TABLE product_type
(
  name VARCHAR(50) NOT NULL PRIMARY KEY
);
CREATE TABLE product_type_count
(
  type  VARCHAR(50)      NOT NULL PRIMARY KEY,
  count INT(10) UNSIGNED NOT NULL DEFAULT 0
);

INSERT INTO product_type (name) VALUES
  ('dress'),
  ('food');

INSERT INTO product (type, name) VALUES
  ('dress', 'T-shirt'),
  ('dress', 'Trousers'),
  ('food', 'Apple'),
  ('food', 'Tomatoes'),
  ('food', 'Meat');
```

We may achieve the goal using stored procedure with using cursor:

```
DELIMITER //
DROP PROCEDURE IF EXISTS product_count;
CREATE PROCEDURE product_count()
  BEGIN
    DECLARE p_type VARCHAR(255);
    DECLARE p_count INT(10) UNSIGNED;
    DECLARE done INT DEFAULT 0;
    DECLARE product CURSOR FOR
      SELECT
        type,
        COUNT(*)
      FROM product
      GROUP BY type;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

    TRUNCATE product_type;

    OPEN product;
```

```
    REPEAT
      FETCH product
      INTO p_type, p_count;
      IF NOT done
      THEN
        INSERT INTO product_type_count
        SET
          type  = p_type,
          count = p_count;
      END IF;
    UNTIL done
    END REPEAT;

    CLOSE product;
  END //
DELIMITER ;
```

When you may call procedure with:

```
CALL product_count();
```

Result would be in `product_type_count` table:

```
type   | count
---------------
dress  |   2
food   |   3
```

While that is a good example of a `CURSOR`, notice how the entire body of the procedure can be replaced by just

```
INSERT INTO product_type_count
        (type, count)
    SELECT type, COUNT(*)
        FROM product
        GROUP BY type;
```

This will run a lot faster.

Section 30.4: Multiple ResultSets

Unlike a `SELECT` statement, a `Stored Procedure` returns multiple result sets. The requires different code to be used for gathering the results of a `CALL` in Perl, PHP, etc.

(Need specific code here or elsewhere!)

Section 30.5: Create a function

```
DELIMITER $$
CREATE
    DEFINER=`db_username`@`hostname_or_IP`
    FUNCTION `function_name`(optional_param data_type(length_if_applicable))
    RETURNS data_type
BEGIN
    /*
    SQL Statements goes here
    */
END$$
```

```
DELIMITER ;
```

The RETURNS data_type is any MySQL datatype.

Chapter 31: Indexes and Keys

Section 31.1: Create index

```
-- Create an index for column 'name' in table 'my_table'
CREATE INDEX idx_name ON my_table(name);
```

Section 31.2: Create unique index

A unique index prevents the insertion of duplicated data in a table. `NULL` values can be inserted in the columns that form part of the unique index (since, by definition, a `NULL` value is different from any other value, including another `NULL` value)

```
-- Creates a unique index for column 'name' in table 'my_table'
CREATE UNIQUE INDEX idx_name ON my_table(name);
```

Section 31.3: AUTO_INCREMENT key

```
CREATE TABLE (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    ...
    PRIMARY KEY(id),
    ...  );
```

Main notes:

- Starts with 1 and increments by 1 automatically when you fail to specify it on `INSERT`, or specify it as `NULL`.
- The ids are always distinct from each other, but...
- Do not make any assumptions (no gaps, consecutively generated, not reused, etc) about the values of the id other than being unique at any given instant.

Subtle notes:

- On restart of server, the 'next' value is 'computed' as `MAX(id)+1`.
- If the last operation before shutdown or crash was to delete the highest id, that id *may* be reused (this is engine-dependent). So, *do not trust auto_increments to be permanently unique*; they are only unique at any moment.
- For multi-master or clustered solutions, see `auto_increment_offset` and `auto_increment_increment`.
- It is OK to have something else as the `PRIMARY KEY` and simply do `INDEX(id)`. (This is an optimization in some situations.)
- Using the `AUTO_INCREMENT` as the "`PARTITION` key" is rarely beneficial; do something different.
- Various operations *may* "burn" values. This happens when they pre-allocate value(s), then don't use them: `INSERT IGNORE` (with dup key), `REPLACE` (which is `DELETE` plus `INSERT`) and others. `ROLLBACK` is another cause for gaps in ids.
- In Replication, you cannot trust ids to arrive at the slave(s) in ascending order. Although ids are assigned in consecutive order, InnoDB statements are sent to slaves in `COMMIT` order.

Section 31.4: Create composite index

This will create a composite index of both keys, `mystring` and `mydatetime` and speed up queries with both columns in the `WHERE` clause.

```
CREATE INDEX idx_mycol_myothercol ON my_table(mycol, myothercol)
```

Note: The order is important! If the search query does not include both columns in the `WHERE` clause, it can only use the leftmost index. In this case, a query with `mycol` in the `WHERE` will use the index, a query searching for `myothercol` **without** also searching for `mycol` will **not**. For more information check out this blog post.

Note: Due to the way BTREE's work, columns that are usually queried in ranges should go in the rightmost value. For example, `DATETIME` columns are usualy queried like `WHERE datecol > '2016-01-01 00:00:00'`. BTREE indexes handle ranges very efficiently but only if the column being queried as a range is the last one in the composite index.

Section 31.5: Drop index

```
-- Drop an index for column 'name' in table 'my_table'
DROP INDEX idx_name ON my_table;
```

Chapter 32: Full-Text search

MySQL offers FULLTEXT searching. It searches tables with columns containing text for the best matches for words and phrases.

Section 32.1: Simple FULLTEXT search

```
   SET @searchTerm= 'Database Programming';
SELECT MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  FROM book
 WHERE MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
 ORDER BY MATCH (Title) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

Given a table named `book` with columns named `ISBN`, 'Title', and 'Author', this finds books matching the terms `'Database Programming'`. It shows the best matches first.

For this to work, a fulltext index on the `Title` column must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

Section 32.2: Simple BOOLEAN search

```
   SET @searchTerm= 'Database Programming -Java';
SELECT MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) Score,
       ISBN, Author, Title
  FROM book
 WHERE MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE)
 ORDER BY MATCH (Title) AGAINST (@searchTerm IN BOOLEAN MODE) DESC;
```

Given a table named `book` with columns named `ISBN`, `Title`, and `Author`, this searches for books with the words `'Database'` and `'Programming'` in the title, but not the word `'Java'`.

For this to work, a fulltext index on the Title column must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_index (Title);
```

Section 32.3: Multi-column FULLTEXT search

```
   SET @searchTerm= 'Date Database Programming';
SELECT MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) Score,
       ISBN, Author, Title
  FROM book
 WHERE MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE)
 ORDER BY MATCH (Title, Author) AGAINST (@searchTerm IN NATURAL LANGUAGE MODE) DESC;
```

Given a table named book with columns named `ISBN`, `Title`, and `Author`, this finds books matching the terms 'Date Database Programming'. It shows the best matches first. The best matches include books written by Prof. C. J. Date.

(But, one of the best matches is also *The Date Doctor's Guide to Dating : How to Get from First Date to Perfect Mate*. This shows up a limitation of FULLTEXT search: it doesn't pretend to understand such things as parts of speech or the meaning of the indexed words.)

For this to work, a fulltext index on the Title and Author columns must be available:

```
ALTER TABLE book ADD FULLTEXT INDEX Fulltext_title_author_index (Title, Author);
```

Chapter 33: PREPARE Statements

Section 33.1: PREPARE, EXECUTE and DEALLOCATE PREPARE Statements

PREPARE prepares a statement for execution

EXECUTE executes a prepared statement

DEALLOCATE PREPARE releases a prepared statement

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt2 FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt2 USING @a, @b;
```

Result:

```
+------------+
| hypotenuse |
+------------+
|         10 |
+------------+
```

Finally,

```
DEALLOCATE PREPARE stmt2;
```

Notes:

- You must use @variables, not DECLAREd variables for `FROM @s`
- A primary use for Prepare, etc, is to 'construct' a query for situations where binding will not work, such as inserting the table name.

Section 33.2: Alter table with add column

```
SET v_column_definition := CONCAT(
  v_column_name
  ,' ',v_column_type
  ,' ',v_column_options
);

SET @stmt := CONCAT('ALTER TABLE ADD COLUMN ', v_column_definition);

PREPARE stmt FROM @stmt;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Chapter 34: JSON

As of MySQL 5.7.8, MySQL supports a native JSON data type that enables efficient access to data in JSON (JavaScript Object Notation) documents. https://dev.mysql.com/doc/refman/5.7/en/json.html

Section 34.1: Create simple table with a primary key and JSON field

```
CREATE TABLE table_name (
    id INT NOT NULL AUTO_INCREMENT,
    json_col JSON,
    PRIMARY KEY(id)
);
```

Section 34.2: Insert a simple JSON

```
INSERT INTO
    table_name (json_col)
VALUES
    ('{"City": "Galle", "Description": "Best damn city in the world"}');
```

That's simple as it can get but note that because JSON dictionary keys have to be surrounded by double quotes the entire thing should be wrapped in single quotes. If the query succeeds, the data will be stored in a binary format.

Section 34.3: Updating a JSON field

In the previous example we saw how mixed data types can be inserted into a JSON field. What if we want to update that field? We are going to add *scheveningen* to the array named `variations` in the previous example.

```
UPDATE
    myjson
SET
    dict=JSON_ARRAY_APPEND(dict,'$.variations','scheveningen')
WHERE
    id = 2;
```

Notes:

1. The `$.variations` array in our json dictionary. The $ symbol represents the json documentation. For a full explaination of json paths recognized by mysql refer to https://dev.mysql.com/doc/refman/5.7/en/json-path-syntax.html
2. Since we don't yet have an example on querying using json fields, this example uses the primary key.

Now if we do `SELECT * FROM myjson` we will see

```
+----+-------------------------------------------------------------------------------------+
| id | dict                                                                                |
+---+--------------------------------------------------------------------------------------+
| 2  | {"opening": "Sicilian", "variations": ["pelikan", "dragon", "najdorf", "scheveningen"]} |
+----+-------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Section 34.4: Insert mixed data into a JSON field

This inserts a json dictionary where one of the members is an array of strings into the table that was created in another example.

```
INSERT INTO myjson(dict)
VALUES('{"opening":"Sicilian","variations":["pelikan","dragon","najdorf"]}');
```

Note, once again, that you need to be careful with the use of single and double quotes. The whole thing has to be wrapped in single quotes.

Section 34.5: CAST data to JSON type

This converts valid json strings to MySQL JSON type:

```
SELECT CAST('[1,2,3]' as JSON) ;
SELECT CAST('{"opening":"Sicilian","variations":["pelikan","dragon","najdorf"]}' as JSON);
```

Section 34.6: Create Json Object and Array

`JSON_OBJECT` creates JSON Objects:

```
SELECT JSON_OBJECT('key1',col1 , 'key2',col2 , 'key3','col3') as myobj;
```

`JSON_ARRAY` creates JSON Array as well:

```
SELECT JSON_ARRAY(col1,col2,'col3') as myarray;
```

Note: myobj.key3 and myarray[2] are "col3" as fixed string.

Also mixed JSON data:

```
SELECT JSON_OBJECT("opening","Sicilian", "variations",JSON_ARRAY("pelikan","dragon","najdorf") )
as mymixed ;
```

Chapter 35: Extract values from JSON type

Parameter	Description
json_doc	valid JSON document
path	members path

MySQL 5.7.8+ supports native JSON type. While you have different ways to create json objects, you can access and read members in different ways, too.

Main function is `JSON_EXTRACT`, hence `->` and `->>` operators are more friendly.

Section 35.1: Read JSON Array value

Create @myjson variable as JSON type (read more):

```
SET @myjson = CAST('["A","B",{"id":1,"label":"C"}]' as JSON) ;
```

`SELECT` some members!

```
SELECT
  JSON_EXTRACT( @myjson , '$[1]' ) ,
  JSON_EXTRACT( @myjson , '$[*].label') ,
  JSON_EXTRACT( @myjson , '$[1].*' ) ,
  JSON_EXTRACT( @myjson , '$[2].*')
;
-- result values:
'\"B\"', '[\"C\"]', NULL, '[1, \"C\"]'
-- visually:
"B", ["C"], NULL, [1, "C"]
```

Section 35.2: JSON Extract Operators

Extract `path` by `->` or `->>` Operators, while `->>` is UNQUOTED value:

```
SELECT
  myjson_col->>'$[1]' , myjson_col->'$[1]' ,
  myjson_col->>'$[*].label' ,
  myjson_col->>'$[1].*' ,
  myjson_col->>'$[2].*'
FROM tablename ;
 -- visuall:
    B, "B" , ["C"], NULL, [1, "C"]
 --^^^ ^^^
```

So `col->>path` is equal to `JSON_UNQUOTE(JSON_EXTRACT(col,path))` :

> As with ->, the ->> operator is always expanded in the output of EXPLAIN, as the following example demonstrates:
>
> ```
> mysql> EXPLAIN SELECT c->>'$.name' AS name
> ->     FROM jemp WHERE g > 2\G
> *************************** 1. row ***************************
> ```

```
id: 1
select_type: SIMPLE
table: jemp
partitions: NULL
type: range
possible_keys: i
key: i
key_len: 5
ref: NULL
rows: 2
filtered: 100.00
Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
Level: Note
Code: 1003
Message: /* select#1 */ select
json_unquote(json_extract(`jtest`.`jemp`.`c`,'$.name')) AS `name` from
`jtest`.`jemp` where (`jtest`.`jemp`.`g` > 2)
1 row in set (0.00 sec)
```

Read about inline path extract(+)

Chapter 36: MySQL Admin

Section 36.1: Atomic RENAME & Table Reload

```
RENAME TABLE t TO t_old, t_copy TO t;
```

No other sessions can access the tables involved while RENAME TABLE executes, so the rename operation is not subject to concurrency problems.

Atomic Rename is especially for completely reloading a table without waiting for `DELETE` and load to finish:

```
CREATE TABLE new LIKE real;
load `new` by whatever means - LOAD DATA, INSERT, whatever
RENAME TABLE real TO old, new TO real;
DROP TABLE old;
```

Section 36.2: Change root password

```
mysqladmin -u root -p'old-password' password 'new-password'
```

Section 36.3: Drop database

Useful for scripting to drop all tables and deletes the database:

```
mysqladmin -u[username] -p[password] drop [database]
```

Use with extreme caution.

To `DROP` database as a SQL Script (you will need DROP privilege on that database):

```
DROP DATABASE database_name
```

or

```
DROP SCHEMA database_name
```

Chapter 37: TRIGGERS

Section 37.1: Basic Trigger

Create Table

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
```

```
Query OK, 0 rows affected (0.03 sec)
```

Create Trigger

```
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
    -> FOR EACH ROW SET @sum = @sum + NEW.amount;
```

```
Query OK, 0 rows affected (0.06 sec)
```

The CREATE TRIGGER statement creates a trigger named ins_sum that is associated with the account table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates

Insert Value

To use the trigger, set the accumulator variable (@sum) to zero, execute an INSERT statement, and then see what value the variable has afterward:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
```

```
+-----------------------+
| Total amount inserted |
+-----------------------+
| 1852.48               |
+-----------------------+
```

In this case, the value of @sum after the INSERT statement has executed is 14.98 + 1937.50 - 100, or 1852.48.

Drop Trigger

```
mysql> DROP TRIGGER test.ins_sum;
```

If you drop a table, any triggers for the table are also dropped.

Section 37.2: Types of triggers

Timing

There are two trigger action time modifiers :

- `BEFORE` trigger activates before executing the request,
- `AFTER` trigger fire after change.

Triggering event

There are three events that triggers can be attached to:

- **INSERT**
- **UPDATE**
- **DELETE**

Before Insert trigger example

```
DELIMITER $$

CREATE TRIGGER insert_date
    BEFORE INSERT ON stack
    FOR EACH ROW
BEGIN
        -- set the insert_date field in the request before the insert
    SET NEW.insert_date = NOW();
END;

$$
DELIMITER ;
```

Before Update trigger example

```
DELIMITER $$

CREATE TRIGGER update_date
    BEFORE UPDATE ON stack
    FOR EACH ROW
BEGIN
        -- set the update_date field in the request before the update
    SET NEW.update_date = NOW();
END;

$$
DELIMITER ;
```

After Delete trigger example

```
DELIMITER $$

CREATE TRIGGER deletion_date
    AFTER DELETE ON stack
    FOR EACH ROW
BEGIN
        -- add a log entry after a successful delete
    INSERT INTO log_action(stack_id, deleted_date) VALUES(OLD.id, NOW());
END;

$$
DELIMITER ;
```

Chapter 38: Configuration and tuning

Section 38.1: InnoDB performance

There are hundreds of settings that can be placed in my.cnf. For the 'lite' user of MySQL, they won't matter as much.

Once your database becomes non-trivial, it is advisable to set the following parameters:

```
innodb_buffer_pool_size
```

This should be set to about 70% of *available* RAM (if you have at least 4GB of RAM; a smaller percentage if you have a tiny VM or antique machine). The setting controls the amount of cache used by the InnoDB ENGINE. Hence, it is very important for performance of InnoDB.

Section 38.2: Parameter to allow huge data to insert

If you need to store images or videos in the column then we need to change the value as needed by your application

max_allowed_packet = 10M

M is Mb, G in Gb, K in Kb

Section 38.3: Increase the string limit for group_concat

`group_concat` is used to concatenate non-null values in a `group`. The maximum length of the resulting string can be set using the `group_concat_max_len` option:

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

Setting the `GLOBAL` variable will ensure a permanent change, whereas setting the `SESSION` variable will set the value for the current session.

Section 38.4: Minimal InnoDB configuration

This is a bare minimum setup for MySQL servers using InnoDB tables. Using InnoDB, query cache is not required. Reclaim disk space when a table or database is `DROP`ed. If you're using SSDs, flushing is a redundant operation (SDDs are not sequential).

```
default_storage_engine = InnoDB
query_cache_type = 0
innodb_file_per_table = 1
innodb_flush_neighbors = 0
```

Concurrency

Make sure we can create more than than the default 4 threads by setting `innodb_thread_concurrency` to infinity (0); this lets InnoDB decide based on optimal execution.

```
innodb_thread_concurrency = 0
innodb_read_io_threads = 64
innodb_write_io_threads = 64
```

Hard drive utilization

Set the capacity (normal load) and capacity_max (absolute maximum) of IOPS for MySQL. The default of 200 is fine for HDDs, but these days, with SSDs capable of thousands of IOPS, you are likely to want to adjust this number. There are many tests you can run to determine IOPS. The values above should be nearly that limit *if you are running a dedicated MySQL server*. If you are running any other services on the same machine, you should apportion as appropriate.

```
innodb_io_capacity = 2500
innodb_io_capacity_max = 3000
```

RAM utilization

Set the RAM available to MySQL. Whilst the rule of thumb is 70-80%, this really depends on whether or not your instance is dedicated to MySQL, and how much RAM is available. Don't *waste* RAM (i.e. resources) if you have a lot available.

```
innodb_buffer_pool_size = 10G
```

Section 38.5: Secure MySQL encryption

The default encryption `aes-128-ecb` uses Electronic Codebook (ECB) mode, which is insecure and should never be used. Instead, add the following to your configuration file:

```
block_encryption_mode = aes-256-cbc
```

Chapter 39: Events

Section 39.1: Create an Event

MySQL has its EVENT functionality for avoiding complicated cron interactions when much of what you are scheduling is SQL related, and less file related. See the Manual page here. Think of Events as Stored Procedures that are scheduled to run on recurring intervals.

To save time in debugging Event-related problems, keep in mind that the global event handler must be turned on to process events.

```
SHOW VARIABLES WHERE variable_name='event_scheduler';
```

```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | OFF   |
+-----------------+-------+
```

With it OFF, nothing will trigger. So turn it on:

```
SET GLOBAL event_scheduler = ON;
```

Schema for testing

```
create table theMessages
(    id INT AUTO_INCREMENT PRIMARY KEY,
    userId INT NOT NULL,
    message VARCHAR(255) NOT NULL,
    updateDt DATETIME NOT NULL,
    KEY(updateDt)
);

INSERT theMessages(userId,message,updateDt) VALUES (1,'message 123','2015-08-24 11:10:09');
INSERT theMessages(userId,message,updateDt) VALUES (7,'message 124','2015-08-29');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 125','2015-09-03 12:00:00');
INSERT theMessages(userId,message,updateDt) VALUES (1,'message 126','2015-09-03 14:00:00');
```

The above inserts are provided to show a starting point. Note that the 2 events created below will clean out rows.

Create 2 events, 1st runs daily, 2nd runs every 10 minutes

Ignore what they are actually doing (playing against one another). The point is on the INTERVAL and scheduling.

```
DROP EVENT IF EXISTS `delete7DayOldMessages`;
DELIMITER $$
CREATE EVENT `delete7DayOldMessages`
  ON SCHEDULE EVERY 1 DAY STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
   DELETE FROM theMessages
   WHERE datediff(now(),updateDt)>6; -- not terribly exact, yesterday but <24hrs is still 1 day

  -- Other code here

END$$
```

```
DELIMITER ;
```

...

```
DROP EVENT IF EXISTS `Every_10_Minutes_Cleanup`;
DELIMITER $$
CREATE EVENT `Every_10_Minutes_Cleanup`
  ON SCHEDULE EVERY 10 MINUTE STARTS '2015-09-01 00:00:00'
  ON COMPLETION PRESERVE
DO BEGIN
   DELETE FROM theMessages
   WHERE TIMESTAMPDIFF(HOUR, updateDt, now())>168; -- messages over 1 week old (168 hours)

   -- Other code here
END$$
DELIMITER ;
```

Show event statuses (different approaches)

```
SHOW EVENTS FROM my_db_name; -- List all events by schema name (db name)
SHOW EVENTS;
SHOW EVENTS\G; -- <--------- I like this one from mysql> prompt
```

```
*************************** 1. row ***************************
Db: my_db_name
Name: delete7DayOldMessages
Definer: root@localhost
Time zone: SYSTEM
Type: RECURRING
Execute at: NULL
Interval value: 1
Interval field: DAY
Starts: 2015-09-01 00:00:00
Ends: NULL
Status: ENABLED
Originator: 1
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: utf8_general_ci
*************************** 2. row ***************************
Db: my_db_name
Name: Every_10_Minutes_Cleanup
Definer: root@localhost
Time zone: SYSTEM
Type: RECURRING
Execute at: NULL
Interval value: 10
Interval field: MINUTE
Starts: 2015-09-01 00:00:00
Ends: NULL
Status: ENABLED
Originator: 1
character_set_client: utf8
collation_connection: utf8_general_ci
Database Collation: utf8_general_ci
2 rows in set (0.06 sec)
```

Random stuff to consider

`DROP EVENT someEventName;` -- Deletes the event and its code

`ON COMPLETION PRESERVE` -- When the event is done processing, retain it. Otherwise, it is deleted.

Events are like triggers. They are not called by a user's program. Rather, they are scheduled. As such, they succeed or fail silently.

The link to the Manual Page shows quite a bit of flexibilty with interval choices, shown below:

> interval:
>
> ```
> quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
>           WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
>           DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
> ```

Events are powerful mechanisms that handle recurring and scheduled tasks for your system. They may contain as many statements, DDL and DML routines, and complicated joins as you may reasonably wish. Please see the MySQL Manual Page entitled Restrictions on Stored Programs.

Chapter 40: ENUM

Section 40.1: Why ENUM?

ENUM provides a way to provide an attribute for a row. Attributes with a small number of non-numeric options work best. Examples:

```
reply ENUM('yes', 'no')
gender ENUM('male', 'female', 'other', 'decline-to-state')
```

The values are strings:

```
INSERT ... VALUES ('yes', 'female')
SELECT ... --> yes female
```

Section 40.2: VARCHAR as an alternative

Let's say we have

```
type ENUM('fish','mammal','bird')
```

An alternative is

```
type VARCHAR(20)  COMENT "fish, bird, etc"
```

This is quite open-ended in that new types are trivially added.

Comparison, and whether better or worse than ENUM:

- (same) INSERT: simply provide the string
- (worse?) On INSERT a typo will go unnoticed
- (same) SELECT: the actual string is returned
- (worse) A lot more space is consumed

Section 40.3: Adding a new option

```
ALTER TABLE tbl MODIFY COLUMN type ENUM('fish','mammal','bird','insect');
```

Notes

- As with all cases of MODIFY COLUMN, you must include `NOT NULL`, and any other qualifiers that originally existed, else they will be lost.
- *If* you add to the *end* of the list *and* the list is under 256 items, the `ALTER` is done by merely changing the schema. That is there will not be a lengthy table copy. (Old versions of MySQL did not have this optimization.)

Section 40.4: NULL vs NOT NULL

Examples of what happens when NULL and 'bad-value' are stored into nullable and not nullable columns. Also shows usage of casting to numeric via `+0`.

```
CREATE TABLE enum (
    e     ENUM('yes', 'no')   NOT NULL,
    enull ENUM('x', 'y', 'z')     NULL
```

```
        );
INSERT INTO enum (e, enull)
    VALUES
        ('yes', 'x'),
        ('no',  'y'),
        (NULL,  NULL),
        ('bad-value', 'bad-value');
```

```
Query OK, 4 rows affected, 3 warnings (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 3
```

```
mysql>SHOW WARNINGS;
```

```
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1048 | Column 'e' cannot be null                 |
| Warning | 1265 | Data truncated for column 'e' at row 4    |
| Warning | 1265 | Data truncated for column 'enull' at row 4 |
+---------+------+-------------------------------------------+
3 rows in set (0.00 sec)
```

What is in the table after those inserts. This uses "+0" to cast to numeric see what is stored.

```
mysql>SELECT e, e+0 FROM enum;
```

```
+-----+-----+
| e   | e+0 |
+-----+-----+
| yes |   1 |
| no  |   2 |
|     |   0 |  -- NULL
|     |   0 |  -- 'bad-value'
+-----+-----+
4 rows in set (0.00 sec)
```

```
mysql>SELECT enull, enull+0 FROM enum;
```

```
+-------+---------+
| enull | enull+0 |
+-------+---------+
| x     |       1 |
| y     |       2 |
| NULL  |    NULL |
|       |       0 |  -- 'bad-value'
+-------+---------+
4 rows in set (0.00 sec)
```

Chapter 41: Install Mysql container with Docker-Compose

Section 41.1: Simple example with docker-compose

This is an simple example to create a mysql server with docker

1.- create **docker-compose.yml**:

Note: If you want to use same container for all your projects, you should create a PATH in your HOME_PATH. If you want to create it for every project you could create a **docker** directory in your project.

```
version: '2'
services:
  cabin_db:
    image: mysql:latest
    volumes:
      - "./.mysql-data/db:/var/lib/mysql"
    restart: always
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: rootpw
      MYSQL_DATABASE: cabin
      MYSQL_USER: cabin
      MYSQL_PASSWORD: cabinpw
```

2.- run it:

```
cd PATH_TO_DOCKER-COMPOSE.YML
docker-compose up -d
```

3.- connect to server

```
mysql -h 127.0.0.1 -u root -P 3306 -p rootpw
```

Hurray!!

4.- stop server

```
docker-compose stop
```

Chapter 42: Character Sets and Collations

Section 42.1: Which CHARACTER SET and COLLATION?

There are dozens of character sets with hundreds of collations. (A given collation belongs to only one character set.) See the output of `SHOW COLLATION;`.

There are usually only 4 `CHARACTER SETs` that matter:

```
ascii -- basic 7-bit codes.
latin1 -- ascii, plus most characters needed for Western European languages.
utf8 -- the 1-, 2-, and 3-byte subset of utf8.  This excludes Emoji and some of Chinese.
utf8mb4 -- the full set of UTF8 characters, covering all current languages.
```

All include English characters, encoded identically. utf8 is a subset of utf8mb4.

Best practice...

- Use utf8mb4 for any `TEXT` or `VARCHAR` column that can have a variety of languages in it.
- Use ascii (latin1 is ok) for hex strings (UUID, MD5, etc) and simple codes (country_code, postal_code, etc).

utf8mb4 did not exist until version 5.5.3, so utf8 was the best available before that.

Outside of MySQL, "UTF8" means the same things as MySQL's utf8mb4, not MySQL's utf8.

Collations start with the charset name and usually end with `_ci` for "case and accent insensitive" or `_bin` for "simply compare the bits.

The 'latest' utf8mb4 collation is `utf8mb4_unicode_520_ci`, based on Unicode 5.20. If you are working with a single language, you might want, say, `utf8mb4_polish_ci`, which will rearrange the letters slightly, based on Polish conventions.

Section 42.2: Setting character sets on tables and fields

You can set a character set both per table, as well as per individual field using the `CHARACTER SET` and `CHARSET` statements:

```
CREATE TABLE Address (
    `AddressID`    INTEGER NOT NULL PRIMARY KEY,
    `Street`       VARCHAR(80) CHARACTER SET ASCII,
    `City`         VARCHAR(80),
    `Country`      VARCHAR(80) DEFAULT "United States",
    `Active`       BOOLEAN DEFAULT 1,
) Engine=InnoDB default charset=UTF8;
```

`City` and `Country` will use `UTF8`, as we set that as the default character set for the table. `Street` on the other hand will use `ASCII`, as we've specifically told it to do so.

Setting the right character set is highly dependent on your dataset, but can also highly improve portability between systems working with your data.

Section 42.3: Declaration

```
CREATE TABLE foo ( ...
    name CHARACTER SET utf8mb4
```

```
    ... );
```

Section 42.4: Connection

Vital to using character sets is to tell the MySQL-server what encoding the client's bytes are. Here is one way:

```
SET NAMES utf8mb4;
```

Each language (PHP, Python, Java, ...) has its own way the it usually preferable to `SET NAMES`.

For example: `SET NAMES utf8mb4`, together with a column declared `CHARACTER SET latin1` -- this will convert from latin1 to utf8mb4 when `INSERTing` and convert back when `SELECTing`.

Chapter 43: MyISAM Engine

Section 43.1: ENGINE=MyISAM

```
CREATE TABLE foo (
      ...
) ENGINE=MyISAM;
```

Chapter 44: Converting from MyISAM to InnoDB

Section 44.1: Basic conversion

```
ALTER TABLE foo ENGINE=InnoDB;
```

This converts the table, but does not take care of any differences between the engines. Most differences will not matter, especially for small tables. But for busier tables, other considerations should be considered. *Conversion considerations*

Section 44.2: Converting All Tables in one Database

To easily convert all tables in one database, use the following:

```
SET @DB_NAME = DATABASE();

SELECT  CONCAT('ALTER TABLE `', table_name, '` ENGINE=InnoDB;') AS sql_statements
FROM    information_schema.tables
WHERE   table_schema = @DB_NAME
AND     `ENGINE` = 'MyISAM'
AND     `TABLE_TYPE` = 'BASE TABLE';
```

> **NOTE:** You should be connected to your database for `DATABASE()` function to work, otherwise it will return `NULL`. This mostly applies to standard mysql client shipped with server as it allows to connect without specifying a database.

Run this SQL statement to retrieve all the `MyISAM` tables in your database.

Finally, copy the output and execute SQL queries from it.

Chapter 45: Transaction

Section 45.1: Start Transaction

A transaction is a sequential group of SQL statements such as select,insert,update or delete, which is performed as one single work unit.

In other words, a transaction will never be complete unless each individual operation within the group is successful. If any operation within the transaction fails, the entire transaction will fail.

Bank transaction will be best example for explaining this. Consider a transfer between two accounts. To achieve this you have to write SQL statements that do the following

1. Check the availability of requested amount in the first account
2. Deduct requested amount from first account
3. Deposit it in second account

If anyone these process fails, the whole should be reverted to their previous state.

ACID : Properties of Transactions

Transactions have the following four standard properties

- **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
- **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** enables transactions to operate independently of and transparent to each other.
- **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Transactions begin with the statement `START TRANSACTION` or `BEGIN WORK` and end with either a `COMMIT` or a `ROLLBACK` statement. The SQL commands between the beginning and ending statements form the bulk of the transaction.

```
START TRANSACTION;
SET @transAmt = '500';
SELECT @availableAmt:=ledgerAmt FROM accTable WHERE customerId=1 FOR UPDATE;
UPDATE accTable SET ledgerAmt=ledgerAmt-@transAmt WHERE customerId=1;
UPDATE accTable SET ledgerAmt=ledgerAmt+@transAmt WHERE customerId=2;
COMMIT;
```

With `START TRANSACTION`, autocommit remains disabled until you end the transaction with `COMMIT` or `ROLLBACK`. The autocommit mode then reverts to its previous state.

The `FOR UPDATE` indicates (and locks) the row(s) for the duration of the transaction.

While the transaction remains uncommitted, this transaction will not be available for others users.

General Procedures involved in Transaction

- Begin transaction by issuing SQL command `BEGIN WORK` or `START TRANSACTION`.
- Run all your SQL statements.
- Check whether everything is executed according to your requirement.
- If yes, then issue `COMMIT` command, otherwise issue a `ROLLBACK` command to revert everything to the previous state.

- Check for errors even after `COMMIT` if you are using, or might eventually use, Galera/PXC.

Section 45.2: COMMIT , ROLLBACK and AUTOCOMMIT

AUTOCOMMIT

MySQL automatically commits statements that are not part of a transaction. The results of any `UPDATE`,`DELETE` or `INSERT` statement not preceded with a `BEGIN` or `START TRANSACTION` will immediately be visible to all connections.

The `AUTOCOMMIT` variable is set *true* by default. This can be changed in the following way,

```
--->To make autcommit false
SET AUTOCOMMIT=false;
--or
SET AUTOCOMMIT=0;

--->To make autcommit true
SET AUTOCOMMIT=true;
--or
SET AUTOCOMMIT=1;
```

To view `AUTOCOMMIT` status

```
SELECT @@autocommit;
```

COMMIT

If `AUTOCOMMIT` set to false and the transaction not committed, the changes will be visible only for the current connection.

After `COMMIT` statement commits the changes to the table, the result will be visible for all connections.

We consider two connections to explain this

Connection 1

```
--->Before making autocommit false one row added in a new table
mysql> INSERT INTO testTable VALUES (1);

--->Making autocommit = false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
+-----+
| tId |
+-----+
|   1 |
|   2 |
|   3 |
+-----+
```

Connection 2

```
mysql> SELECT * FROM testTable;
+-----+
| tId |
```

```
+-----+
|   1 |
+-----+
---> Row inserted before autocommit=false only visible here
```

Connection 1

```
mysql> COMMIT;
--->Now COMMIT is executed in connection 1
mysql> SELECT * FROM testTable;
    +-----+
    | tId |
    +-----+
    |   1 |
    |   2 |
    |   3 |
    +-----+
```

Connection 2

```
mysql> SELECT * FROM testTable;
    +-----+
    | tId |
    +-----+
    |   1 |
    |   2 |
    |   3 |
    +-----+
--->Now all the three rows are visible here
```

ROLLBACK

If anything went wrong in your query execution, **ROLLBACK** in used to revert the changes. See the explanation below

```
--->Before making autocommit false one row added in a new table
mysql> INSERT INTO testTable VALUES (1);

--->Making autocommit = false
mysql> SET autocommit=0;

mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
+-----+
| tId |
+-----+
|   1 |
|   2 |
|   3 |
+-----+
```

Now we are executing **ROLLBACK**

```
--->Rollback executed now
mysql> ROLLBACk;

mysql> SELECT * FROM testTable;
+-----+
| tId |
+-----+
```

```
|   1 |
+-----+
--->Rollback removed all rows which all are not committed
```

Once `COMMIT` is executed, then `ROLLBACK` will not cause anything

```
mysql> INSERT INTO testTable VALUES (2), (3);
mysql> SELECT * FROM testTable;
mysql> COMMIT;
+-----+
| tId |
+-----+
|   1 |
|   2 |
|   3 |
+-----+

--->Rollback executed now
mysql> ROLLBACk;

mysql> SELECT * FROM testTable;
+-----+
| tId |
+-----+
|   1 |
|   2 |
|   3 |
+-----+
--->Rollback not removed any rows
```

If `AUTOCOMMIT` is set *true*, then `COMMIT` and `ROLLBACK` is useless

Section 45.3: Transaction using JDBC Driver

Transaction using JDBC driver is used to control how and when a transaction should commit and rollback. Connection to MySQL server is created using JDBC driver

JDBC driver for MySQL can be downloaded here

Lets start with getting a connection to database using JDBC driver

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);
--->Example for connection url "jdbc:mysql://localhost:3306/testDB");
```

Character Sets : This indicates what character set the client will use to send SQL statements to the server. It also specifies the character set that the server should use for sending results back to the client.

This should be mentioned while creating connection to server. So the connection string should be like,

```
jdbc:mysql://localhost:3306/testDB?useUnicode=true&amp;characterEncoding=utf8
```

See this for more details about Character Sets and Collations

When you open connection, the `AUTOCOMMIT` mode is set to *true* by default, that should be changed *false* to start transaction.

```
con.setAutoCommit(false);
```

You should always call `setAutoCommit()` method right after you open a connection.

Otherwise use **`START TRANSACTION`** or **`BEGIN WORK`** to start a new transaction. By using **`START TRANSACTION`** or **`BEGIN WORK`**, no need to change `AUTOCOMMIT` *false*. That will be automatically disabled.

Now you can start transaction. See a complete JDBC transaction example below.

```
package jdbcTest;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;


public class accTrans {

    public static void doTransfer(double transAmount,int customerIdFrom,int customerIdTo) {

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            String DB_CONNECTION_URL =
"jdbc:mysql://localhost:3306/testDB?useUnicode=true&amp;characterEncoding=utf8";

            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(DB_CONNECTION_URL,DB_USER,USER_PASSWORD);

            --->set auto commit to false
            con.setAutoCommit(false);
            ---> or use con.START TRANSACTION / con.BEGIN WORK

            --->Start SQL Statements for transaction
            --->Checking availability of amount
            double availableAmt    = 0;
            pstmt = con.prepareStatement("SELECT ledgerAmt FROM accTable WHERE customerId=? FOR
UPDATE");
            pstmt.setInt(1, customerIdFrom);
            rs = pstmt.executeQuery();
            if(rs.next())
                availableAmt    = rs.getDouble(1);

            if(availableAmt >= transAmount)
            {
                ---> Do Transfer
                ---> taking amount from cutomerIdFrom
                pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt-? WHERE
customerId=?");
                pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdFrom);
                pstmt.executeUpdate();

                ---> depositing amount in cutomerIdTo
                pstmt = con.prepareStatement("UPDATE accTable SET ledgerAmt=ledgerAmt+? WHERE
customerId=?");
                pstmt.setDouble(1, transAmount);
                pstmt.setInt(2, customerIdTo);
                pstmt.executeUpdate();
```

```
                con.commit();
            }
            --->If you performed any insert,update or delete operations before
            ----> this availability check, then include this else part
            /*else { --->Rollback the transaction if availability is less than required
                con.rollback();
            }*/

        } catch (SQLException ex) {
            ---> Rollback the transaction in case of any error
            con.rollback();
        } finally {
            try {
                if(rs != null)  rs.close();
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            }
        }
    }

    public static void main(String[] args) {
        doTransfer(500, 1020, 1021);
        -->doTransfer(transAmount, customerIdFrom, customerIdTo);
    }
}
```

JDBC transaction make sure of all SQL statements within a transaction block are executed successful, if either one of the SQL statement within transaction block is failed, abort and rollback everything within the transaction block.

Chapter 46: Log files

Section 46.1: Slow Query Log

The Slow Query Log consists of log events for queries taking up to `long_query_time` seconds to finish. For instance, up to 10 seconds to complete. To see the time threshold currently set, issue the following:

```
SELECT @@long_query_time;
+-------------------+
| @@long_query_time |
+-------------------+
|         10.000000 |
+-------------------+
```

It can be set as a GLOBAL variable, in `my.cnf` or `my.ini` file. Or it can be set by the connection, though this is unusual. The value can be set between 0 to 10 (seconds). What value to use?

- 10 is so high as to be almost useless;
- 2 is a compromise;
- 0.5 and other fractions are possible;
- 0 captures everything; this could fill up disk dangerously fast, but can be very useful.

The capturing of slow queries is either turned on or off. And the file logged to is also specified. The below captures these concepts:

```
SELECT @@slow_query_log; -- Is capture currently active? (1=On, 0=Off)
SELECT @@slow_query_log_file; -- filename for capture. Resides in datadir
SELECT @@datadir; -- to see current value of the location for capture file

SET GLOBAL slow_query_log=0; -- Turn Off
-- make a backup of the Slow Query Log capture file. Then delete it.
SET GLOBAL slow_query_log=1; -- Turn it back On (new empty file is created)
```

For more information, please see the MySQL Manual Page The Slow Query Log

Note: The above information on turning on/off the slowlog was changed in 5.6(?); older version had another mechanism.

The "best" way to see what is slowing down your system:

```
long_query_time=...
turn on the slowlog
run for a few hours
turn off the slowlog (or raise the cutoff)
run pt-query-digest to find the 'worst' couple of queries.  Or mysqldumpslow -s t
```

Section 46.2: A List

- General log - all queries - see VARIABLE general_log
- Slow log - queries slower than long_query_time - slow_query_log_file
- Binlog - for replication and backup - log_bin_basename
- Relay log - also for replication
- general errors - mysqld.err

- start/stop - mysql.log (not very interesting) - log_error
- InnoDB redo log - iblog*

See the variables `basedir` and `datadir` for default location for many logs

Some logs are turned on/off by other VARIABLES. Some are either written to a file or to a table.

(Note to reviewers: This needs more details and more explanation.)

Documenters: please include the default location and name for each log type, for both Windows and *nix. (Or at least as much as you can.)

Section 46.3: General Query Log

The General Query Log contains a listing of general information from client connects, disconnects, and queries. It is invaluable for debugging, yet it poses as a hindrance to performance (citation?).

An example view of a General Query Log is seen below:

To determine if the General Log is currently being captured:

```
SELECT @@general_log; -- 1 = Capture is active; 0 = It is not.
```

To determine the filename of the capture file:

```
SELECT @@general_log_file; -- Full path to capture file
```

If the fullpath to the file is not shown, the file exists in the `datadir`.

Windows example:

```
+-----------------------------------------------------------+
| @@general_log_file                                        |
+-----------------------------------------------------------+
| C:\ProgramData\MySQL\MySQL Server 5.7\Data\GuySmiley.log |
+-----------------------------------------------------------+
```

Linux:

```
+----------------------------------+
| @@general_log_file               |
+----------------------------------+
| /var/lib/mysql/ip-ww-xx-yy-zz.log |
+----------------------------------+
```

When changes are made to the `general_log_file` GLOBAL variable, the new log is saved in the `datadir`. However, the fullpath may no longer be reflected by examining the variable.

In the case of no entry for `general_log_file` in the configuration file, it will default to `@@hostname`.log in the `datadir`.

Best practices are to turn OFF capture. Save the log file to a backup directory with a filename reflecting the begin/end datetime of the capture. Deleting the prior file if a filesystem *move* did not occur of that file. Establish a new filename for the log file and turn capture ON (all show below). Best practices also include a careful determination if you even want to capture at the moment. Typically, capture is ON for debugging purposes only.

A typical filesystem filename for a backed-up log might be:

```
/LogBackup/GeneralLog_20160802_1520_to_20160802_1815.log
```

where the date and time are part to the filename as a range.

For Windows note the following sequence with setting changes.

```
SELECT @@general_log; -- 0. Not being captured
SELECT @@general_log_file; -- C:\ProgramData\MySQL\MySQL Server 5.6\Data\GuySmiley.log
SELECT @@datadir; -- C:\ProgramData\MySQL\MySQL Server 5.7\Data\
SET GLOBAL general_log_file='GeneralLogBegin_20160803_1420.log'; -- datetime clue
SET GLOBAL general_log=1; -- Turns on actual log capture. File is created under `datadir`
SET GLOBAL general_log=0; -- Turn logging off
```

Linux is similar. These would represent dynamic changes. Any restart of the server would pick up configuration file settings.

As for the configuration file, consider the following relevant variable settings:

```
[mysqld]
general_log_file = /path/to/currentquery.log
general_log      = 1
```

In addition, the variable `log_output` can be configured for `TABLE` output, not just `FILE`. For that, please see Destinations.

Please see the MySQL Manual Page The General Query Log.

Section 46.4: Error Log

The Error Log is populated with start and stop information, and critical events encountered by the server.

The following is an example of its contents:

The variable `log_error` holds the path to the log file for error logging.

In the absence of a configuration file entry for `log_error`, the system will default its values to `@@hostname`.err in the `datadir`. Note that `log_error` is not a dynamic variable. As such, changes are done through a cnf or ini file changes and a server restart (or by seeing "Flushing and Renaming the Error Log File" in the Manual Page link at the bottom here).

Logging cannot be disabled for errors. They are important for system health while troubleshooting problems. Also, entries are infrequent compared to the General Query Log.

The GLOBAL variable `log_warnings` sets the level for verbosity which varies by server version. The following snippet illustrates:

```
SELECT @@log_warnings; -- make a note of your prior setting
SET GLOBAL log_warnings=2; -- setting above 1 increases output (see server version)
```

`log_warnings` as seen above is a dynamic variable.

Configuration file changes in `cnf` and `ini` files might look like the following.

```
[mysqld]
log_error        = /path/to/CurrentError.log
log_warnings     = 2
```

MySQL 5.7.2 expanded the warning level verbosity to 3 and added the GLOBAL `log_error_verbosity`. Again, it was introduced in 5.7.2. It can be set dynamically and checked as a variable or set via `cnf` or `ini` configuration file settings.

As of MySQL 5.7.2:

```
[mysqld]
log_error           = /path/to/CurrentError.log
log_warnings        = 2
log_error_verbosity = 3
```

Please see the MySQL Manual Page entitled The Error Log especially for Flushing and Renaming the Error Log file, and its *Error Log Verbosity* section with versions related to `log_warnings` and `error_log_verbosity`.

Chapter 47: Clustering

Section 47.1: Disambiguation

"MySQL Cluster" disambiguation...

- NDB Cluster -- A specialized, mostly in-memory, engine. Not widely used.
- Galera Cluster aka Percona XtraDB Cluster aka PXC aka MariaDB with Galera. -- A very good High Availability solution for MySQL; it goes beyond Replication.

See individual pages on those variants of "Cluster".

For "clustered index" see page(s) on `PRIMARY KEY`.

Chapter 48: Partitioning

Section 48.1: RANGE Partitioning

A table that is partitioned by range is partitioned in such a way that each partition contains rows for which the partitioning expression value lies within a given range. Ranges should be contiguous but not overlapping, and are defined using the `VALUES LESS THAN` operator. For the next few examples, suppose that you are creating a table such as the following to hold personnel records for a chain of 20 video stores, numbered 1 through 20:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
);
```

This table can be partitioned by range in a number of ways, depending on your needs. One way would be to use the `store_id` column. For instance, you might decide to partition the table 4 ways by adding a `PARTITION BY RANGE` clause as shown here:

```
ALTER TABLE employees PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

> `MAXVALUE` represents an integer value that is always greater than the largest possible integer value (in mathematical language, it serves as a least upper bound).

based on MySQL official document.

Section 48.2: LIST Partitioning

List partitioning is similar to range partitioning in many ways. As in partitioning by RANGE, each partition must be explicitly defined. The chief difference between the two types of partitioning is that, in list partitioning, each partition is defined and selected based on the membership of a column value in one of a set of value lists, rather than in one of a set of contiguous ranges of values. This is done by using `PARTITION BY LIST(expr)` where `expr` is a column value or an expression based on a column value and returning an integer value, and then defining each partition by means of a `VALUES IN (value_list)`, where `value_list` is a comma-separated list of integers.

For the examples that follow, we assume that the basic definition of the table to be partitioned is provided by the `CREATE TABLE` statement shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
```

```
    job_code INT,
    store_id INT
);
```

Suppose that there are 20 video stores distributed among 4 franchises as shown in the following table.

Region	Store ID Numbers
North	3, 5, 6, 9, 17
East	1, 2, 10, 11, 19, 20
West	4, 12, 13, 14, 18
Central	7, 8, 15, 16

To partition this table in such a way that rows for stores belonging to the same region are stored in the same partition

```
ALTER TABLE employees PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

based on MySQL official document.

Section 48.3: HASH Partitioning

Partitioning by HASH is used primarily to ensure an even distribution of data among a predetermined number of partitions. With range or list partitioning, you must specify explicitly into which partition a given column value or set of column values is to be stored; with hash partitioning, MySQL takes care of this for you, and you need only specify a column value or expression based on a column value to be hashed and the number of partitions into which the partitioned table is to be divided.

The following statement creates a table that uses hashing on the store_id column and is divided into 4 partitions:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

> If you do not include a `PARTITIONS` clause, the number of partitions defaults to 1.

based on MySQL official document.

Chapter 49: Replication

Section 49.1: Master - Slave Replication Setup

Consider 2 MySQL Servers for replication setup, one is a Master and the other is a Slave.

We are going to configure the Master that it should keep a log of every action performed on it. We are going to configure the Slave server that it should look at the log on the Master and whenever changes happens in log on the Master, it should do the same thing.

Master Configuration

First of all, we need to create a user on the Master. This user is going to be used by Slave to create a connection with the Master.

```
CREATE USER 'user_name'@'%' IDENTIFIED BY 'user_password';
GRANT REPLICATION SLAVE ON *.* TO 'user_name'@'%';
FLUSH PRIVILEGES;
```

Change `user_name` and `user_password` according to your Username and Password.

Now `my.inf` (my.cnf in Linux) file should be edited. Include the following lines in [mysqld] section.

```
server-id = 1
log-bin = mysql-bin.log
binlog-do-db = your_database
```

The first line is used to assign an ID to this MySQL server.

The second line tells MySQL to start writing a log in the specified log file. In Linux this can be configured like `log-bin = /home/mysql/logs/mysql-bin.log`. If you are starting replication in a MySQL server in which replication has already been used, make sure this directory is empty of all replication logs.

The third line is used to configure the database for which we are going to write log. You should replace `your_database` with your database name.

Make sure `skip-networking` has not been enabled and restart the MySQL server(Master)

Slave Configuration

`my.inf` file should be edited in Slave also. Include the following lines in [mysqld] section.

```
server-id = 2
master-host = master_ip_address
master-connect-retry = 60

master-user = user_name
master-password = user_password
replicate-do-db = your_database

relay-log = slave-relay.log
relay-log-index = slave-relay-log.index
```

The first line is used to assign an ID to this MySQL server. This ID should be unique.

The second line is the I.P address of the Master server. Change this according to your Master system I.P.

The third line is used to set a retry limit in seconds.

The next two lines tell the username and password to the Slave, by using which it connect the Master.

Next line set the database it needs to replicate.

The last two lines used to assign `relay-log` and `relay-log-index` file names.

Make sure `skip-networking` has not been enabled and restart the MySQL server(Slave)

Copy Data to Slave

If data is constantly being added to the Master, we will have to prevent all database access on the Master so nothing can be added. This can be achieved by run the following statement in Master.

```
FLUSH TABLES WITH READ LOCK;
```

If no data is being added to the server, you can skip the above step.

We are going to take data backup of the Master by using `mysqldump`

```
mysqldump your_database -u root -p > D://Backup/backup.sql;
```

Change `your_database` and backup directory according to your setup. You wll now have a file called `backup.sql` in the given location.

If your database not exists in your Slave, create that by executing the following

```
CREATE DATABASE `your_database`;
```

Now we have to import backup into Slave MySQL server.

```
mysql -u root -p your_database  <D://Backup/backup.sql
--->Change `your_database` and backup directory according to your setup
```

Start Replication

To start replication, we need to find the log file name and log position in the Master. So, run the following in Master

```
SHOW MASTER STATUS;
```

This will give you an output like below

```
+--------------------+----------+--------------------------------+------------------+
| File               | Position | Binlog_Do_DB                   | Binlog_Ignore_DB |
+--------------------+----------+--------------------------------+------------------+
| mysql-bin.000001   | 130      | your_database                  |                  |
+--------------------+----------+--------------------------------+------------------+
```

Then run the following in Slave

```
SLAVE STOP;
CHANGE MASTER TO MASTER_HOST='master_ip_address', MASTER_USER='user_name',
```

```
   MASTER_PASSWORD='user_password', MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=130;
SLAVE START;
```

First we stop the Slave. Then we tell it exactly where to look in the Master log file. For `MASTER_LOG_FILE` name and `MASTER_LOG_POS`, use the values which we got by running `SHOW MASTER STATUS` command on the Master.

You should change the I.P of the Master in `MASTER_HOST`, and change the user and password accordingly.

The Slave will now be waiting. The status of the Slave can be viewed by run the following

```
SHOW SLAVE STATUS;
```

If you previously executed `FLUSH TABLES WITH READ LOCK` in Master, release the tables from lock by run the following

```
UNLOCK TABLES;
```

Now the Master keep a log for every action performed on it and the Slave server look at the log on the Master. Whenever changes happens in log on the Master, Slave replicate that.

Section 49.2: Replication Errors

Whenever there is an error while running a query on the slave, MySQL stop replication automatically to identify the problem and fix it. This mainly because an event caused a duplicate key or a row was not found and it cannot be updated or deleted. You can skip such errors, even if this is not recommended

To skip just one query that is hanging the slave, use the following syntax

```
SET GLOBAL sql_slave_skip_counter = N;
```

This statement skips the next N events from the master. This statement is valid only when the slave threads are not running. Otherwise, it produces an error.

```
STOP SLAVE;
SET GLOBAL sql_slave_skip_counter=1;
START SLAVE;
```

In some cases this is fine. But if the statement is part of a multi-statement transaction, it becomes more complex, because skipping the error producing statement will cause the whole transaction to be skipped.

If you want to skip more queries which producing same error code and if you are sure that skipping those errors will not bring your slave inconsistent and you want to skip them all, you would add a line to skip that error code in your `my.cnf`.

For example you might want to skip all duplicate errors you might be getting

```
1062 | Error 'Duplicate entry 'xyz' for key 1' on query
```

Then add the following to your `my.cnf`

```
slave-skip-errors = 1062
```

You can skip also other type of errors or all error codes, but make sure that skipping those errors will not bring your slave inconsistent. The following are the syntax and examples

```
slave-skip-errors=[err_code1,err_code2,...|all]

slave-skip-errors=1062,1053
slave-skip-errors=all
slave-skip-errors=ddl_exist_errors
```

Chapter 50: Backup using mysqldump

Option	Effect
--	# Server login options
`-h` (`--host`)	Host (IP address or hostname) to connect to. Default is `localhost` (`127.0.0.1`) Example: `-h localhost`
`-u` (`--user`)	MySQL user
`-p` (`--password`)	MySQL password. **Important**: When using `-p`, there must not be a space between the option and the password. Example: `-pMyPassword`
--	# Dump options
`--add-drop-database`	Add a `DROP DATABASE` statement before each `CREATE DATABASE` statement. Useful if you want to replace databases in the server.
`--add-drop-table`	Add a `DROP TABLE` statement before each `CREATE TABLE` statement. Useful if you want to replace tables in the server.
`--no-create-db`	Suppress the `CREATE DATABASE` statements in the dump. This is useful when you're sure the database(s) you're dumping already exist(s) in the server where you'll load the dump.
`-t` (`--no-create-info`)	Suppress all `CREATE TABLE` statements in the dump. This is useful when you want to dump only the data from the tables and will use the dump file to populate identical tables in another database / server.
`-d` (`--no-data`)	Do not write table information. This will only dump the `CREATE TABLE` statements. Useful for creating "template" databases
`-R` (`--routines`)	Include stored procedures / functions in the dump.
`-K` (`--disable-keys`)	Disable keys for each table before inserting the data, and enable keys after the data is inserted. This speeds up inserts only in MyISAM tables with non-unique indexes.

Section 50.1: Specifying username and password

```
> mysqldump -u username -p [other options]
Enter password:
```

If you need to specify the password on the command line (e.g. in a script), you can add it after the `-p` option *without* a space:

```
> mysqldump -u username -ppassword [other options]
```

If you password contains spaces or special characters, remember to use escaping depending on your shell / system.

Optionally the extended form is:

```
> mysqldump --user=username --password=password [other options]
```

(Explicity specifying the password on the commandline is Not Recommended due to security concerns.)

Section 50.2: Creating a backup of a database or table

Create a snapshot of a whole database:

```
mysqldump [options] db_name > filename.sql
```

Create a snapshot of multiple databases:

```
mysqldump [options] --databases db_name1 db_name2 ... > filename.sql
```

```
mysqldump [options] --all-databases > filename.sql
```

Create a snapshot of one or more tables:

```
mysqldump [options] db_name table_name... > filename.sql
```

Create a snapshot *excluding* one or more tables:

```
mysqldump [options] db_name --ignore-table=tbl1 --ignore-table=tbl2 ... > filename.sql
```

The file extension `.sql` is fully a matter of style. Any extension would work.

Section 50.3: Restoring a backup of a database or table

```
mysql [options] db_name < filename.sql
```

Note that:

- `db_name` needs to be an existing database;
- your authenticated user has sufficient privileges to execute all the commands inside your `filename.sql`;
- The file extension `.sql` is fully a matter of style. Any extension would work.
- You cannot specify a table name to load into even though you could specify one to dump from. This must be done within `filename.sql`.

Alternatively, when in the **MySQL Command line tool**, you can restore (or run any other script) by using the source command:

```
source filename.sql
```

or

```
\. filename.sql
```

Section 50.4: Tranferring data from one MySQL server to another

If you need to copy a database from one server to another, you have two options:

Option 1:

1. Store the dump file in the source server
2. Copy the dump file to your destination server
3. Load the dump file into your destination server

On the source server:

```
mysqldump [options] > dump.sql
```

On the destination server, copy the dump file and execute:

```
mysql [options] < dump.sql
```

Option 2:

If the destination server can connect to the host server, you can use a pipeline to copy the database from one server to the other:

On the destination server

```
mysqldump [options to connect to the source server] | mysql [options]
```

Similarly, the script could be run on the source server, pushing to the destination. In either case, it is likely to be significantly faster than Option 1.

Section 50.5: mysqldump from a remote server with compression

In order to use compression over the wire for a faster transfer, pass the `--compress` option to `mysqldump`. Example:

```
mysqldump -h db.example.com -u username -p --compress dbname > dbname.sql
```

Important: If you don't want to lock up the *source* db, you should also include `--lock-tables=false`. But you may not get an internally consistent db image that way.

To also save the file compressed, you can pipe to `gzip`.

```
mysqldump -h db.example.com -u username -p --compress dbname | gzip --stdout > dbname.sql.gz
```

Section 50.6: restore a gzipped mysqldump file without uncompressing

```
gunzip -c dbname.sql.gz | mysql dbname -u username -p
```

Note: `-c` means write output to stdout.

Section 50.7: Backup database with stored procedures and functions

By default stored procedures and functions or not generated by `mysqldump`, you will need to add the parameter `--routines` (or `-R`):

```
mysqldump -u username -p -R db_name > dump.sql
```

When using `--routines` the creation and change time stamps are not maintained, instead you should dump and reload the contents of `mysql.proc`.

Section 50.8: Backup direct to Amazon S3 with compression

If you wish to make a complete backup of a large MySql installation and do not have sufficient local storage, you can dump and compress it directly to an Amazon S3 bucket. It's also a good practice to do this without having the DB password as part of the command:

```
mysqldump -u root -p --host=localhost --opt --skip-lock-tables --single-transaction \
        --verbose --hex-blob --routines --triggers --all-databases |
    gzip -9 | s3cmd put - s3://s3-bucket/db-server-name.sql.gz
```

You are prompted for the password, after which the backup starts.

Chapter 51: mysqlimport

Parameter	Description
`--delete -D`	empty the table before importing the text file
`--fields-optionally-enclosed-by`	define the character that quotes the fields
`--fields-terminated-by`	field terminator
`--ignore -i`	ignore the ingested row in case of duplicate-keys
`--lines-terminated-by`	define row terminator
`--password -p`	password
`--port -P`	port
`--replace -r`	overwrite the old entry row in case of duplicate-keys
`--user -u`	username
`--where -w`	specify a condition

Section 51.1: Basic usage

Given the tab-separated file `employee.txt`

> 1 \t Arthur Dent
> 2 \t Marvin
> 3 \t Zaphod Beeblebrox

```
$ mysql --user=user --password=password mycompany -e 'CREATE TABLE employee(id INT, name
VARCHAR(100), PRIMARY KEY (id))'

$ mysqlimport --user=user --password=password mycompany employee.txt
```

Section 51.2: Using a custom field-delimiter

Given the text file employee.txt

> 1|Arthur Dent
> 2|Marvin
> 3|Zaphod Beeblebrox

```
$ mysqlimport --fields-terminated-by='|' mycompany employee.txt
```

Section 51.3: Using a custom row-delimiter

This example is useful for windows-like endings:

```
$ mysqlimport --lines-terminated-by='\r\n' mycompany employee.txt
```

Section 51.4: Handling duplicate keys

Given the table `Employee`

id	Name

3 Yooden Vranx

And the file `employee.txt`

> 1 \t Arthur Dent
> 2 \t Marvin
> 3 \t Zaphod Beeblebrox

The `--ignore` option will ignore the entry on duplicate keys

```
$ mysqlimport --ignore mycompany employee.txt
```

id	Name
1	Arthur Dent
2	Marvin
3	Yooden Vranx

The `--replace` option will overwrite the old entry

```
$ mysqlimport --replace mycompany employee.txt
```

id	Name
1	Arthur Dent
2	Marvin
3	Zaphod Beeblebrox

Section 51.5: Conditional import

```
$ mysqlimport --where="id>2" mycompany employee.txt
```

Section 51.6: Import a standard csv

```
$ mysqlimport
    --fields-optionally-enclosed-by='"'
    --fields-terminated-by=,
    --lines-terminated-by="\r\n"
    mycompany employee.csv
```

Section 52.2: Load data with duplicates

If you use the `LOAD DATA INFILE` command to populate a table with existing data, you will often find that the import fails due to duplicates. There are several possible ways to overcome this problem.

LOAD DATA LOCAL

If this option has been enabled in your server, it can be used to load a file that exists on the client computer rather than the server. A side effect is that duplicate rows for unique values are ignored.

```
LOAD DATA LOCAL INFILE 'path of the file/file_name.txt'
INTO TABLE employee
```

LOAD DATA INFILE 'fname' REPLACE

When the replace keyword is used duplicate unique or primary keys will result in the existing row being replaced with new ones

```
LOAD DATA INFILE 'path of the file/file_name.txt'
REPLACE INTO TABLE employee
```

LOAD DATA INFILE 'fname' IGNORE

The opposite of `REPLACE`, existing rows will be preserved and new ones ignored. This behavior is similar to `LOCAL` described above. However the file need not exist on the client computer.

```
LOAD DATA INFILE 'path of the file/file_name.txt'
IGNORE INTO TABLE employee
```

Load via intermediary table

Sometimes ignoring or replacing all duplicates may not be the ideal option. You may need to make decisions based on the contents of other columns. In that case the best option is to load into an intermediary table and transfer from there.

```
INSERT INTO employee SELECT * FROM intermediary WHERE ...
```

Section 52.3: Import a CSV file into a MySQL table

The following command imports CSV files into a MySQL table with the same columns while respecting CSV quoting and escaping rules.

```
load data infile '/tmp/file.csv'
into table my_table
fields terminated by ','
optionally enclosed by '"'
escaped by '"'
lines terminated by '\n'
ignore 1 lines; -- skip the header row
```

Chapter 53: MySQL Unions

Section 53.1: Union operator

The UNION operator is used to combine the result-set (*only distinct values*) of two or more SELECT statements.

Query: (To selects all the different cities (*only distinct values*) from the "Customers" and the "Suppliers" tables)

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

Result:

```
Number of Records: 10

City
------
Aachen
Albuquerque
Anchorage
Annecy
Barcelona
Barquisimeto
Bend
Bergamo
Berlin
Bern
```

Section 53.2: Union ALL

UNION ALL to select all (duplicate values also) cities from the "Customers" and "Suppliers" tables.

Query:

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

Result:

```
Number of Records: 12

City
-------
Aachen
Albuquerque
Anchorage
Ann Arbor
Annecy
Barcelona
Barquisimeto
Bend
Bergamo
```

```
Berlin
Berlin
Bern
```

Section 53.3: UNION ALL With WHERE

UNION ALL to select all(duplicate values also) German cities from the "Customers" and "Suppliers" tables. Here `Country="Germany"` is to be specified in the where clause.

Query:

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

Result:

```
Number of Records: 14
```

City	Country
Aachen	Germany
Berlin	Germany
Berlin	Germany
Brandenburg	Germany
Cunewalde	Germany
Cuxhaven	Germany
Frankfurt	Germany
Frankfurt a.M.	Germany
Köln	Germany
Leipzig	Germany
Mannheim	Germany
München	Germany
Münster	Germany
Stuttgart	Germany

Chapter 54: MySQL client

Parameter	Description
`-D --database=name`	name of the database
`--delimiter=str`	set the statement delimiter. The default one is ';'
`-e --execute='command'`	execute command
`-h --host=name`	hostname to connect to
`-p --password=name`	password *Note: there is no space between `-p` and the password*
`-p` (without password)	the password will be prompted for
`-P --port=#`	port number
`-s --silent`	silent mode, produce less output. Use `\t` as column separator
`-ss`	like `-s`, but omit column names
`-S --socket=path`	specify the socket (Unix) or named pipe (Windows) to use when connecting to a local instance
`--skip-column-names`	omit column names
`-u --user=name`	username
`-U --safe-updates --i-am-a-dummy`	login with the variable `sql_safe_updates=ON`. This will allow only **`DELETE`** and **`UPDATE`** that explicitly use keys
`-V --version`	print the version and exit

Section 54.1: Base login

To access MySQL from the command line:

```
mysql --user=username --password=pwd --host=hostname test_db
```

This can be shortened to:

```
mysql -u username -p password -h hostname test_db
```

By omitting the `password` value MySQL will ask for any required password as the first input. If you specify `password` the client will give you an 'insecure' warning:

```
mysql -u=username -p -h=hostname test_db
```

For local connections `--socket` can be used to point to the socket file:

```
mysql --user=username --password=pwd --host=localhost --socket=/path/to/mysqld.sock test_db
```

Omitting the `socket` parameter will cause the client to attempt to attach to a server on the local machine. The server must be running to connect to it.

Section 54.2: Execute commands

This set of example show how to execute commands stored in strings or script files, without the need of the interactive prompt. This is especially useful to when a shell script needs to interact with a database.

Execute command from a string

```
$ mysql -uroot -proot test -e'select * from people'
```

```
+----+-------+--------+
| id | name  | gender |
+----+-------+--------+
|  1 | Kathy | f      |
|  2 | John  | m      |
+----+-------+--------+
```

To format the output as a tab-separated grid, use the `--silent` parameter:

```
$ mysql -uroot -proot test -s -e'select * from people'

id      name    gender
1       Kathy   f
2       John    m
```

To omit the headers:

```
$ mysql -uroot -proot test -ss -e'select * from people'

1       Kathy   f
2       John    m
```

Execute from script file:

```
$ mysql -uroot -proot test < my_script.sql
```

```
$ mysql -uroot -proot test -e'source my_script.sql'
```

Write the output on a file

```
$ mysql -uroot -proot test < my_script.sql > out.txt

$ mysql -uroot -proot test -s -e'select * from people' > out.txt
```

Chapter 55: Temporary Tables

Section 55.1: Create Temporary Table

Temporary tables could be very useful to keep temporary data. Temporary tables option is available in MySQL version 3.23 and above.

Temporary table will be automatically destroyed when the session ends or connection is closed. The user can also drop temporary table.

Same temporary table name can be used in many connections at the same time, because the temporary table is only available and accessible by the client who creates that table.

The temporary table can be created in the following types

```
--->Basic temporary table creation
CREATE TEMPORARY TABLE tempTable1(
       id INT NOT NULL AUTO_INCREMENT,
       title VARCHAR(100) NOT NULL,
       PRIMARY KEY ( id )
    );

--->Temporary table creation from select query
CREATE TEMPORARY TABLE tempTable1
    SELECT ColumnName1,ColumnName2,... FROM table1;
```

You can add indexes as you build the table:

```
CREATE TEMPORARY TABLE tempTable1
        ( PRIMARY KEY(ColumnName2) )
    SELECT ColumnName1,ColumnName2,... FROM table1;
```

`IF NOT EXISTS` key word can be used as mentioned below to avoid *'table already exists'* error. But in that case table will not be created, if the table name which you are using already exists in your current session.

```
CREATE TEMPORARY TABLE IF NOT EXISTS tempTable1
    SELECT ColumnName1,ColumnName2,... FROM table1;
```

Section 55.2: Drop Temporary Table

Drop Temporary Table is used to delete the temporary table which you are created in your current session.

```
DROP TEMPORARY TABLE tempTable1

DROP TEMPORARY TABLE IF EXISTS tempTable1
```

Use `IF EXISTS` to prevent an error occurring for tables that may not exist

Chapter 56: Customize PS1

Section 56.1: Customize the MySQL PS1 with current database

In the .bashrc or .bash_profile, adding:

```
export MYSQL_PS1="\u@\h [\d]>"
```

make the MySQL client PROMPT show current user@host [database].

Section 56.2: Custom PS1 via MySQL configuration file

In `mysqld.cnf` or equivalent:

```
[mysql]
prompt = '\u@\h [\d]> '
```

This achieves a similar effect, without having to deal with `.bashrc`'s.

Chapter 57: Dealing with sparse or missing data

Section 57.1: Working with columns containg NULL values

In MySQL and other SQL dialects, `NULL` values have special properties.

Consider the following table containing job applicants, the companies they worked for, and the date they left the company. `NULL` indicates that an applicant still works at the company:

```
CREATE TABLE example
(`applicant_id` INT, `company_name` VARCHAR(255), `end_date` DATE);
```

```
+--------------+-----------------+------------+
| applicant_id | company_name    | end_date   |
+--------------+-----------------+------------+
|            1 | Google          | NULL       |
|            1 | Initech         | 2013-01-31 |
|            2 | Woodworking.com | 2016-08-25 |
|            2 | NY Times        | 2013-11-10 |
|            3 | NFL.com         | 2014-04-13 |
+--------------+-----------------+------------+
```

Your task is to compose a query that returns all rows after `2016-01-01`, including any employees that are still working at a company (those with `NULL` end dates). This select statement:

```
SELECT * FROM example WHERE end_date > '2016-01-01';
```

fails to include any rows with `NULL` values:

```
+--------------+-----------------+------------+
| applicant_id | company_name    | end_date   |
+--------------+-----------------+------------+
|            2 | Woodworking.com | 2016-08-25 |
+--------------+-----------------+------------+
```

Per the MySQL documentation, comparisons using the arithmetic operators <, >, =, and <> themselves return `NULL` instead of a boolean `TRUE` or `FALSE`. Thus a row with a `NULL` end_date is neither greater than 2016-01-01 nor less than 2016-01-01.

This can be solved by using the keywords IS NULL:

```
SELECT * FROM example WHERE end_date > '2016-01-01' OR end_date IS NULL;
```

```
+--------------+-----------------+------------+
| applicant_id | company_name    | end_date   |
+--------------+-----------------+------------+
|            1 | Google          | NULL       |
|            2 | Woodworking.com | 2016-08-25 |
+--------------+-----------------+------------+
```

Working with NULLs becomes more complex when the task involves aggregation functions like `MAX()` and a `GROUP`

BY clause. If your task were to select the most recent employed date for each applicant_id, the following query would seem a logical first attempt:

```
SELECT applicant_id, MAX(end_date) FROM example GROUP BY applicant_id;
```

```
+--------------+---------------+
| applicant_id | MAX(end_date) |
+--------------+---------------+
|            1 | 2013-01-31    |
|            2 | 2016-08-25    |
|            3 | 2014-04-13    |
+--------------+---------------+
```

However, knowing that `NULL` indicates an applicant is still employed at a company, the first row of the result is inaccurate. Using `CASE WHEN` provides a workaround for the `NULL` issue:

```
 SELECT
    applicant_id,
    CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END
    max_date
FROM example
GROUP BY applicant_id;
```

```
+--------------+------------+
| applicant_id | max_date   |
+--------------+------------+
|            1 | present    |
|            2 | 2016-08-25 |
|            3 | 2014-04-13 |
+--------------+------------+
```

This result can be joined back to the original `example` table to determine the company at which an applicant last worked:

```
SELECT
  data.applicant_id,
  data.company_name,
  data.max_date
FROM (
  SELECT
    *,
    CASE WHEN end_date is null THEN 'present' ELSE end_date END max_date
  FROM example
) data
INNER JOIN (
 SELECT
   applicant_id,
   CASE WHEN MAX(end_date is null) = 1 THEN 'present' ELSE MAX(end_date) END max_date
 FROM
   example
 GROUP BY applicant_id
) j
ON data.applicant_id = j.applicant_id AND data.max_date = j.max_date;
```

```
+--------------+-----------------+------------+
| applicant_id | company_name    | max_date   |
+--------------+-----------------+------------+
```

```
|            1 | Google          | present    |
|            2 | Woodworking.com | 2016-08-25 |
|            3 | NFL.com         | 2014-04-13 |
+--------------+-----------------+------------+
```

These are just a few examples of working with **NULL** values in MySQL.

Chapter 58: Connecting with UTF-8 Using Various Programming language.

Section 58.1: Python

1st or 2nd line in source code (to have literals in the code utf8-encoded):

```
# -*- coding: utf-8 -*-
```

Connection:

```
db = MySQLdb.connect(host=DB_HOST, user=DB_USER, passwd=DB_PASS, db=DB_NAME,
        charset="utf8mb4", use_unicode=True)
```

For web pages, one of these:

```
<meta charset="utf-8" />
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Section 58.2: PHP

In php.ini (this is the default after PHP 5.6):

```
default_charset UTF-8
```

When building a web page:

```
header('Content-type: text/plain; charset=UTF-8');
```

When connecting to MySQL:

```
(for mysql:)   Do not use the mysql_* API!
(for mysqli:)  $mysqli_obj->set_charset('utf8mb4');
(for PDO:)     $db = new PDO('dblib:host=host;dbname=db;charset=utf8', $user, $pwd);
```

In code, do not use any conversion routines.

For data entry,

```
<form accept-charset="UTF-8">
```

For JSON, to avoid \uxxxx:

```
$t = json_encode($s, JSON_UNESCAPED_UNICODE);
```

Chapter 59: Time with subsecond precision

Section 59.1: Get the current time with millisecond precision

```
SELECT NOW(3)
```

does the trick.

Section 59.2: Get the current time in a form that looks like a Javascript timestamp

Javascript timestamps are based on the venerable UNIX `time_t` data type, and show the number of milliseconds since `1970-01-01 00:00:00` UTC.

This expression gets the current time as a Javascript timestamp integer. (It does so correctly regardless of the current time_zone setting.)

```
ROUND(UNIX_TIMESTAMP(NOW(3)) * 1000.0, 0)
```

If you have `TIMESTAMP` values stored in a column, you can retrieve them as integer Javascript timestamps using the UNIX_TIMESTAMP() function.

```
  SELECT ROUND(UNIX_TIMESTAMP(column) * 1000.0, 0)
```

If your column contains `DATETIME` columns and you retrieve them as Javascript timestamps, those timestamps will be offset by the time zone offset of the time zone they're stored in.

Section 59.3: Create a table with columns to store sub-second time

```
CREATE TABLE times (
     dt DATETIME(3),
     ts TIMESTAMP(3)
 );
```

makes a table with millisecond-precision date / time fields.

```
INSERT INTO times VALUES (NOW(3), NOW(3));
```

inserts a row containing `NOW()` values with millisecond precision into the table.

```
INSERT INTO times VALUES ('2015-01-01 16:34:00.123','2015-01-01 16:34:00.128');
```

inserts specific millisecond precision values.

Notice that you must use `NOW(3)` rather than `NOW()` if you use that function to insert high-precision time values.

Section 59.4: Convert a millisecond-precision date / time value to text

`%f` is the fractional precision format specifier for the DATE_FORMAT() function.

```
SELECT DATE_FORMAT(NOW(3), '%Y-%m-%d %H:%i:%s.%f')
```

displays a value like `2016-11-19 09:52:53.248000` with fractional microseconds. Because we used `NOW(3)`, the final three digits in the fraction are 0.

Section 59.5: Store a Javascript timestamp into a TIMESTAMP column

If you have a Javascript timestamp value, for example `1478960868932`, you can convert that to a MySQL fractional time value like this:

```
FROM_UNIXTIME(1478960868932 * 0.001)
```

It's simple to use that kind of expression to store your Javascript timestamp into a MySQL table. Do this:

```
INSERT INTO table (col) VALUES (FROM_UNIXTIME(1478960868932 * 0.001))
```

(Obviously, you'll want to insert other columns.)

Chapter 60: One to Many

The idea of one to many (1:M) concerns the joining of rows to each other, specifically cases where a single row in one table corresponds to many rows in another.

1:M is one-directional, that is, any time you query a 1:M relationship, you can use the 'one' row to select 'many' rows in another table, but you cannot use a single 'many' row to select more than a single 'one' row.

Section 60.1: Example Company Tables

Consider a company where every employee who is a manager, manages 1 or more employees, and every employee has only 1 manager.

This results in two tables:

EMPLOYEES

EMP_ID	FIRST_NAME	LAST_NAME	MGR_ID
E01	Johnny	Appleseed	M02
E02	Erin	Macklemore	M01
E03	Colby	Paperwork	M03
E04	Ron	Sonswan	M01

MANAGERS

MGR_ID	FIRST_NAME	LAST_NAME
M01	Loud	McQueen
M02	Bossy	Pants
M03	Barrel	Jones

Section 60.2: Get the Employees Managed by a Single Manager

```
SELECT e.emp_id , e.first_name , e.last_name FROM employees e INNER JOIN managers m ON m.mgr_id =
e.mgr_id WHERE m.mgr_id = 'M01' ;
```

Results in:

EMP_ID	FIRST_NAME	LAST_NAME
E02	Erin	Macklemore
E04	Ron	Sonswan

Ultimately, for every manager we query for, we will see 1 or more employees returned.

Section 60.3: Get the Manager for a Single Employee

Consult the above example tables when looking at this example.

```
SELECT m.mgr_id , m.first_name , m.last_name FROM managers m INNER JOIN employees e ON e.mgr_id =
m.mgr_id WHERE e.emp_id = 'E03' ;
```

MGR_ID	FIRST_NAME	LAST_NAME
M03	Barrel	Jones

As this is the inverse of the above example, we know that for every employee we query for, we will only ever see one corresponding manager.

Chapter 61: Server Information

Parameters	Explanation
GLOBAL	Shows the variables as they are configured for the entire server. Optional.
SESSION	Shows the variables that are configured for this session only. Optional.

Section 61.1: SHOW VARIABLES example

To get all the server variables run this query either in the SQL window of your preferred interface (PHPMyAdmin or other) or in the MySQL CLI interface

```
SHOW VARIABLES;
```

You can specify if you want the session variables or the global variables as follows:

Session variables:

```
SHOW SESSION VARIABLES;
```

Global variables:

```
SHOW GLOBAL VARIABLES;
```

Like any other SQL command you can add parameters to your query such as the LIKE command:

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE 'max_join_size';
```

Or, using wildcards:

```
SHOW [GLOBAL | SESSION] VARIABLES LIKE '%size%';
```

You can also filter the results of the SHOW query using a WHERE parameter as follows:

```
SHOW [GLOBAL | SESSION] VARIABLES WHERE VALUE > 0;
```

Section 61.2: SHOW STATUS example

To get the database server status run this query in either the SQL window of your preferred interface (PHPMyAdmin or other) or on the MySQL CLI interface.

```
SHOW STATUS;
```

You can specify whether you wish to receive the SESSION or GLOBAL status of your sever like so: Session status:

```
SHOW SESSION STATUS;
```

Global status:

```
SHOW GLOBAL STATUS;
```

Like any other SQL command you can add parameters to your query such as the LIKE command:

```
SHOW [GLOBAL | SESSION] STATUS LIKE 'Key%';
```

Or the Where command:

```
SHOW [GLOBAL | SESSION] STATUS WHERE VALUE > 0;
```

The main difference between GLOBAL and SESSION is that with the GLOBAL modifier the command displays aggregated information about the server and all of it's connections, while the SESSION modifier will only show the values for the current connection.

Chapter 62: SSL Connection Setup

Section 62.1: Setup for Debian-based systems

(This assumes MySQL has been installed and that `sudo` is being used.)

Generating a CA and SSL keys

Make sure OpenSSL and libraries are installed:

```
apt-get -y install openssl
apt-get -y install libssl-dev
```

Next make and enter a directory for the SSL files:

```
mkdir /home/ubuntu/mysqlcerts
cd /home/ubuntu/mysqlcerts
```

To generate keys, create a certificate authority (CA) to sign the keys (self-signed):

```
openssl genrsa 2048 > ca-key.pem
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca.pem
```

The values entered at each prompt won't affect the configuration. Next create a key for the server, and sign using the CA from before:

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out server-req.pem
openssl rsa -in server-key.pem -out server-key.pem

openssl x509 -req -in server-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out
server-cert.pem
```

Then create a key for a client:

```
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout client-key.pem -out client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -req -in client-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -set_serial 01 -out
client-cert.pem
```

To make sure everything was set up correctly, verify the keys:

```
openssl verify -CAfile ca.pem server-cert.pem client-cert.pem
```

Adding the keys to MySQL

Open the MySQL configuration file. For example:

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

Under the `[mysqld]` section, add the following options:

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/server-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/server-key.pem
```

Restart MySQL. For example:

```
service mysql restart
```

Test the SSL connection

Connect in the same way, passing in the extra options `ssl-ca`, `ssl-cert`, and `ssl-key`, using the generated client key. For example, assuming `cd /home/ubuntu/mysqlcerts`:

```
mysql --ssl-ca=ca.pem --ssl-cert=client-cert.pem --ssl-key=client-key.pem -h 127.0.0.1 -u superman
-p
```

After logging in, verify the connection is indeed secure:

```
superman@127.0.0.1 [None]> SHOW VARIABLES LIKE '%ssl%';
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| have_openssl  | YES                                     |
| have_ssl      | YES                                     |
| ssl_ca        | /home/ubuntu/mysqlcerts/ca.pem          |
| ssl_capath    |                                         |
| ssl_cert      | /home/ubuntu/mysqlcerts/server-cert.pem |
| ssl_cipher    |                                         |
| ssl_crl       |                                         |
| ssl_crlpath   |                                         |
| ssl_key       | /home/ubuntu/mysqlcerts/server-key.pem  |
+---------------+-----------------------------------------+
```

You could also check:

```
superman@127.0.0.1 [None]> STATUS;
...
SSL:                    Cipher in use is DHE-RSA-AES256-SHA
...
```

Enforcing SSL

This is via `GRANT`, using `REQUIRE SSL`:

```
GRANT ALL PRIVILEGES ON *.* TO 'superman'@'127.0.0.1' IDENTIFIED BY 'pass' REQUIRE SSL;
FLUSH PRIVILEGES;
```

Now, `superman` *must* connect via SSL.

If you don't want to manage client keys, use the client key from earlier and automatically use that for all clients. Open MySQL configuration file, for example:

```
vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

Under the `[client]` section, add the following options:

```
ssl-ca = /home/ubuntu/mysqlcerts/ca.pem
ssl-cert = /home/ubuntu/mysqlcerts/client-cert.pem
ssl-key = /home/ubuntu/mysqlcerts/client-key.pem
```

Now `superman` only has to type the following to login via SSL:

```
mysql -h 127.0.0.1 -u superman -p
```

Connecting from another program, for example in Python, typically only requires an additional parameter to the connect function. A Python example:

```
import MySQLdb
ssl = {'cert': '/home/ubuntu/mysqlcerts/client-cert.pem', 'key': '/home/ubuntu/mysqlcerts/client-key.pem'}
conn = MySQLdb.connect(host='127.0.0.1', user='superman', passwd='imsoawesome', ssl=ssl)
```

References and further reading:

- https://www.percona.com/blog/2013/06/22/setting-up-mysql-ssl-and-secure-connections/
- https://lowendbox.com/blog/getting-started-with-mysql-over-ssl/
- http://xmodulo.com/enable-ssl-mysql-server-client.html
- https://ubuntuforums.org/showthread.php?t=1121458

Section 62.2: Setup for CentOS7 / RHEL7

This example assumes two servers:

1. dbserver (where our database lives)
2. appclient (where our applications live)

FWIW, both servers are SELinux enforcing.

First, log on to dbserver

Create a temporary directory for creating the certificates.

```
mkdir /root/certs/mysql/ && cd /root/certs/mysql/
```

Create the server certificates

```
openssl genrsa 2048 > ca-key.pem
openssl req -sha1 -new -x509 -nodes -days 3650 -key ca-key.pem > ca-cert.pem
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout server-key.pem > server-req.pem
openssl rsa -in server-key.pem -out server-key.pem
openssl x509 -sha1 -req -in server-req.pem -days 730  -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 > server-cert.pem
```

Move server certificates to /etc/pki/tls/certs/mysql/

Directory path assumes CentOS or RHEL (adjust as needed for other distros):

```
mkdir /etc/pki/tls/certs/mysql/
```

Be sure to set permissions on the folder and files. mysql needs full ownership and access.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

Now configure MySQL/MariaDB

```
# vi /etc/my.cnf
# i
```

```
[mysqld]
bind-address=*
ssl-ca=/etc/pki/tls/certs/ca-cert.pem
ssl-cert=/etc/pki/tls/certs/server-cert.pem
ssl-key=/etc/pki/tls/certs/server-key.pem
# :wq
```

Then

```
systemctl restart mariadb
```

Don't forget to open your firewall to allow connections from appclient (using IP 1.2.3.4)

```
firewall-cmd --zone=drop --permanent --add-rich-rule 'rule family="ipv4" source address="1.2.3.4"
service name="mysql" accept'
# I force everything to the drop zone.  Season the above command to taste.
```

Now restart firewalld

```
service firewalld restart
```

Next, log in to dbserver's mysql server:

```
mysql -uroot -p
```

Issue the following to create a user for the client. note REQUIRE SSL in GRANT statement.

```
GRANT ALL PRIVILEGES ON *.* TO 'iamsecure'@'appclient' IDENTIFIED BY 'dingdingding' REQUIRE SSL;
FLUSH PRIVILEGES;
# quit mysql
```

You should still be in /root/certs/mysql from the first step. If not, cd back to it for one of the commands below.

Create the client certificates

```
openssl req -sha1 -newkey rsa:2048 -days 730 -nodes -keyout client-key.pem > client-req.pem
openssl rsa -in client-key.pem -out client-key.pem
openssl x509 -sha1 -req -in client-req.pem -days 730 -CA ca-cert.pem -CAkey ca-key.pem -set_serial
01 > client-cert.pem
```

Note: I used the same common name for both server and client certificates. YMMV.

Be sure you're still /root/certs/mysql/ for this next command

Combine server and client CA certificate into a single file:

```
cat server-cert.pem client-cert.pem > ca.pem
```

Make sure you see two certificates:

```
cat ca.pem
```

END OF SERVER SIDE WORK FOR NOW.

Open another terminal and

```
ssh appclient
```

As before, create a permanent home for the client certificates

```
mkdir /etc/pki/tls/certs/mysql/
```

Now, place the client certificates (created on dbserver) on appclient. You can either scp them over, or just copy and paste the files one by one.

```
scp dbserver
# copy files from dbserver to appclient
# exit scp
```

Again, be sure to set permissions on the folder and files. mysql needs full ownership and access.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

You should have three files, each owned by user mysql:

```
/etc/pki/tls/certs/mysql/ca.pem
/etc/pki/tls/certs/mysql/client-cert.pem
/etc/pki/tls/certs/mysql/client-key.pem
```

Now edit appclient's MariaDB/MySQL config in the `[client]` section.

```
vi /etc/my.cnf
# i
[client]
ssl-ca=/etc/pki/tls/certs/mysql/ca.pem
ssl-cert=/etc/pki/tls/certs/mysql/client-cert.pem
ssl-key=/etc/pki/tls/certs/mysql/client-key.pem
# :wq
```

Restart appclient's mariadb service:

```
systemctl restart mariadb
```

still on the client here

This should return: ssl TRUE

```
mysql --ssl --help
```

Now, log in to appclient's mysql instance

```
mysql -uroot -p
```

Should see YES to both variables below

```
show variables LIKE '%ssl';
    have_openssl    YES
    have_ssl              YES
```

Initially I saw

```
 have_openssl NO
```

A quick look into mariadb.log revealed:

> SSL error: Unable to get certificate from '/etc/pki/tls/certs/mysql/client-cert.pem'

The problem was that root owned client-cert.pem and the containing folder. The solution was to set ownership of /etc/pki/tls/certs/mysql/ to mysql.

```
chown -R mysql:mysql /etc/pki/tls/certs/mysql
```

Restart mariadb if needed from the step immediately above

NOW WE ARE READY TO TEST THE SECURE CONNECTION
We're still on appclient here

Attempt to connect to dbserver's mysql instance using the account created above.

```
mysql -h dbserver -u iamsecure -p
# enter password dingdingding (hopefully you changed that to something else)
```

With a little luck you should be logged in without error.

To confirm you are connected with SSL enabled, issue the following command from the MariaDB/MySQL prompt:

```
\s
```

That's a backslash s, aka status

That will show the status of your connection, which should look something like this:

```
Connection id:        4
Current database:
Current user:        iamsecure@appclient
SSL:            Cipher in use is DHE-RSA-AES256-GCM-SHA384
Current pager:        stdout
Using outfile:        ''
Using delimiter:    ;
Server:            MariaDB
Server version:        5.X.X-MariaDB MariaDB Server
Protocol version:    10
Connection:        dbserver via TCP/IP
Server characterset:    latin1
Db     characterset:    latin1
Client characterset:    utf8
Conn.  characterset:    utf8
TCP port:        3306
Uptime:            42 min 13 sec
```

If you get permission denied errors on your connection attempt, check your GRANT statement above to make sure there aren't any stray characters or ' marks.

If you have SSL errors, go back through this guide to make sure the steps are orderly.

This worked on RHEL7 and will likely work on CentOS7, too. Cannot confirm whether these exact steps will work elsewhere.

Hope this saves someone else a little time and aggravation.

Chapter 63: Create New User

Section 63.1: Create a MySQL User

For creating new user, We need to follow simple steps as below :

Step 1: Login to `MySQL` as `root`

```
$ mysql -u root -p
```

Step 2 : We will see mysql command prompt

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

Here, We have successfully created new user, But this user won't have any `permissions`, So to assign `permissions` to user use following command :

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO 'my_new_user'@'localhost' identified by 'my_password';
```

Section 63.2: Specify the password

The basic usage is:

```
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY 'test_password';
```

However for situations where is not advisable to hard-code the password in cleartext it is also possible to specify directly, using the directive `PASSWORD`, the hashed value as returned by the `PASSWORD()` function:

```
mysql> select PASSWORD('test_password'); -- returns *4414E26EDED6D661B5386813EBBA95065DBC4728
mysql> CREATE USER 'my_new_user'@'localhost' IDENTIFIED BY PASSWORD
'*4414E26EDED6D661B5386813EBBA95065DBC4728';
```

Section 63.3: Create new user and grant all priviliges to schema

```
grant all privileges on schema_name.* to 'new_user_name'@'%' identified by 'newpassword';
```

Attention: This can be used to create new root user

Section 63.4: Renaming user

```
rename user 'user'@'%' to 'new_name`@'%';
```

If you create a user by mistake, you can change his name

Chapter 64: Security via GRANTs

Section 64.1: Best Practice

Limit root (and any other SUPER-privileged user) to

```
GRANT ... TO root@localhost ...
```

That prevents access from other servers. You should hand out SUPER to very few people, and they should be aware of their responsibility. The application should not have SUPER.

Limit application logins to the one database it uses:

```
GRANT ... ON dbname.* ...
```

That way, someone who hacks into the application code can't get past dbname. This can be further refined via either of these:

```
GRANT SELECT ON dname.* ...    -- "read only"
GRANT ... ON dname.tblname ... -- "just one table"
```

The readonly may also need 'safe' things like

```
GRANT SELECT, CREATE TEMPORARY TABLE ON dname.* ...    -- "read only"
```

As you say, there is no absolute security. My point here is there you can do a few things to slow hackers down. (Same goes for honest people goofing.)

In rare cases, you may need the application to do something available only to root. this can be done via a "Stored Procedure" that has `SECURITY DEFINER` (and root defines it). That will expose only what the SP does, which might, for example, be one particular action on one particular table.

Section 64.2: Host (of user@host)

The "host" can be either a host name or an IP address. Also, it can involve wild cards.

```
GRANT SELECT ON db.* TO sam@'my.domain.com' IDENTIFIED BY 'foo';
```

Examples: Note: these usually need to be quoted

```
localhost -- the same machine as mysqld
'my.domain.com' -- a specific domain; this involves a lookup
'11.22.33.44' -- a specific IP address
'192.168.1.%' -- wild card for trailing part of IP address.  (192.168.% and 10.% and 11.% are
"internal" ip addresses.)
```

Using `localhost` relies on the security of the server. For best practice `root` should only be allowed in through localhost. In some cases, these mean the same thing: `0.0.0.1` and `::1`.

Chapter 65: Change Password

Section 65.1: Change MySQL root password in Linux

To change MySQL's root user password:

Step 1: Stop the MySQL server.

- in Ubuntu or Debian:
  ```
  sudo /etc/init.d/mysql stop
  ```
- in CentOS, Fedora or Red Hat Enterprise Linux:
  ```
  sudo /etc/init.d/mysqld stop
  ```

Step 2: Start the MySQL server without the privilege system.

```
sudo mysqld_safe --skip-grant-tables &
```

or, if `mysqld_safe` is unavailable,

```
sudo mysqld --skip-grant-tables &
```

Step 3: Connect to the MySQL server.

```
mysql -u root
```

Step 4: Set a new password for root user.

Version > 5.7

```
FLUSH PRIVILEGES;
ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';
FLUSH PRIVILEGES;
exit;
```

Version ≤ 5.7

```
FLUSH PRIVILEGES;
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new_password');
FLUSH PRIVILEGES;
exit;
```

Note: The `ALTER USER` syntax was introduced in MySQL 5.7.6.

Step 5: Restart the MySQL server.

- in Ubuntu or Debian:
  ```
  sudo /etc/init.d/mysql stop
  sudo /etc/init.d/mysql start
  ```
- in CentOS, Fedora or Red Hat Enterprise Linux:
  ```
  sudo /etc/init.d/mysqld stop
  sudo /etc/init.d/mysqld start
  ```

Section 65.2: Change MySQL root password in Windows

When we want to change root password in windows, We need to follow following steps :

Step 1 : Start your Command Prompt by using any of below method :

Perss `Crtl+R` or Goto **`Start`** `Menu > Run` and then type `cmd` and hit enter

Step 2 : Change your directory to where `MYSQL` is installed, In my case it's

```
C:\> cd C:\mysql\bin
```

Step 3 : Now we need to start `mysql` command prompt

```
C:\mysql\bin> mysql -u root mysql
```

Step 4 : Fire query to change `root` password

```
mysql> SET PASSWORD FOR root@localhost=PASSWORD('my_new_password');
```

Section 65.3: Process

1. Stop the MySQL (mysqld) server/daemon process.
2. Start the MySQL server process the --skip-grant-tables option so that it will not prompt for a password: `mysqld_safe --skip-grant-tables &`
3. Connect to the MySQL server as the root user: `mysql -u root`
4. Change password:

- (5.7.6 and newer): `ALTER USER 'root'@'localhost' IDENTIFIED BY 'new-password';`
- (5.7.5 and older, or MariaDB): `SET PASSWORD FOR 'root'@'localhost' = PASSWORD('new-password); flush privileges; quit;`

5. Restart the MySQL server.

Note: this will work only if you are physically on the same server.

Online Doc: http://dev.mysql.com/doc/refman/5.7/en/resetting-permissions.html

Chapter 66: Recover and reset the default root password for MySQL 5.7+

After MySQL 5.7, when we install MySQL sometimes we don't need to create a root account or give a root password. By default when we start the server, the default password is stored in the `mysqld.log` file. We need to login in to the system using that password and we need to change it.

Section 66.1: What happens when the initial start up of the server

Given that the data directory of the server is empty:

- The server is initialized.
- SSL certificate and key files are generated in the data directory.
- The validate_password plugin is installed and enabled.
- The superuser account 'root'@'localhost' is created. The password for the superuser is set and stored in the error log file.

Section 66.2: How to change the root password by using the default password

To reveal the default "root" password:

```
shell> sudo grep 'temporary password' /var/log/mysqld.log
```

Change the root password as soon as possible by logging in with the generated temporary password and set a custom password for the superuser account:

```
shell> mysql -uroot -p

mysql> ALTER USER 'root'@'localhost' IDENTIFIED BY 'MyNewPass5!';
```

Note: MySQL's validate_password plugin is installed by default. This will require that passwords contain at least one upper case letter, one lower case letter, one digit, and one special character, and that the total password length is at least 8 characters.

Section 66.3: reset root password when " /var/run/mysqld' for UNIX socket file don't exists"

if I forget the password then I'll get error.

```
$ mysql -u root -p
```

Enter password:

ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)

I tried to solve the issue by first knowing the status:

```
$ systemctl status mysql.service
```

mysql.service - MySQL Community Server Loaded: loaded (/lib/systemd/system/mysql.service; enabled; vendor

preset: en Active: active (running) since Thu 2017-06-08 14:31:33 IST; 38s ago

Then I used the code `mysqld_safe --skip-grant-tables &` but I get the error:

> mysqld_safe Directory '/var/run/mysqld' for UNIX socket file don't exists.

```
$ systemctl stop  mysql.service
$ ps -eaf|grep mysql
$ mysqld_safe --skip-grant-tables &
```

> I solved:

```
$ mkdir -p /var/run/mysqld
$ chown mysql:mysql /var/run/mysqld
```

Now I use the same code `mysqld_safe --skip-grant-tables &` and get

> mysqld_safe Starting mysqld daemon with databases from /var/lib/mysql

If I use `$ mysql -u root` I'll get :

Server version: 5.7.18-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

> Now time to change password:

```
mysql> use mysql
mysql> describe user;
```

Reading table information for completion of table and column names You can turn off this feature to get a quicker startup with -A

Database changed

```
mysql> FLUSH PRIVILEGES;
mysql> SET PASSWORD FOR root@'localhost' = PASSWORD('newpwd');
```

or If you have a mysql root account that can connect from everywhere, you should also do:

```
UPDATE mysql.user SET Password=PASSWORD('newpwd') WHERE User='root';
```

Alternate Method:

```
  USE mysql
  UPDATE user SET Password = PASSWORD('newpwd')
  WHERE Host = 'localhost' AND User = 'root';
```

And if you have a root account that can access from everywhere:

```
USE mysql
UPDATE user SET Password = PASSWORD('newpwd')
WHERE Host = '%' AND User = 'root';`enter code here
```

now need to `quit` from mysql and stop/start

```
FLUSH PRIVILEGES;
sudo /etc/init.d/mysql stop
sudo /etc/init.d/mysql start
```

now again ` mysql -u root -p' and use the new password to get

> mysql>

Chapter 67: Recover from lost root password

Section 67.1: Set root password, enable root user for socket and http access

Solves problem of: access denied for user root using password YES Stop mySQL:

```
sudo systemctl stop mysql
```

Restart mySQL, skipping grant tables:

```
sudo mysqld_safe --skip-grant-tables
```

Login:

```
mysql -u root
```

In SQL shell, look if users exist:

```
select User, password,plugin FROM mysql.user ;
```

Update the users (plugin null enables for all plugins):

```
update mysql.user set password=PASSWORD('mypassword'), plugin = NULL WHERE User = 'root';
exit;
```

In Unix shell stop mySQL without grant tables, then restart with grant tables:

```
sudo service mysql stop
sudo service mysql start
```

Chapter 68: MySQL Performance Tips

Section 68.1: Building a composite index

In many situations, a composite index performs better than an index with a single column. To build an optimal composite index, populate it with columns in this order.

- = column(s) from the `WHERE` clause first. (eg, `INDEX(a,b,...)` for `WHERE a=12 AND b='xyz' ...`)
- `IN` column(s); the optimizer may be able to leapfrog through the index.
- One "range" (eg `x BETWEEN 3 AND 9`, `name LIKE 'J%'`) It won't use anything past the first range column.
- All the columns in `GROUP BY`, in order
- All the columns in `ORDER BY`, in order. Works only if all are `ASC` or all are `DESC` or you are using 8.0.

Notes and exceptions:

- Don't duplicate any columns.
- Skip over any cases that don't apply.
- If you don't use all the columns of `WHERE`, there is no need to go on to `GROUP BY`, etc.
- There are cases where it is useful to index only the `ORDER BY` column(s), ignoring `WHERE`.
- Don't "hide" a column in a function (eg `DATE(x) = ...` cannot use x in the index.)
- 'Prefix' indexing (eg, `text_col(99)`) is unlikely to be helpful; may hurt.

More details and tips .

Section 68.2: Optimizing Storage Layout for InnoDB Tables

1. In InnoDB, having a long PRIMARY KEY (either a single column with a lengthy value, or several columns that form a long composite value) wastes a lot of disk space. The primary key value for a row is duplicated in all the secondary index records that point to the same row. Create an AUTO_INCREMENT column as the primary key if your primary key is long.
2. Use the VARCHAR data type instead of CHAR to store variable-length strings or for columns with many NULL values. A CHAR(N) column always takes N characters to store data, even if the string is shorter or its value is NULL. Smaller tables fit better in the buffer pool and reduce disk I/O.

> When using COMPACT row format (the default InnoDB format) and variable-length character sets, such as utf8 or sjis, CHAR(N) columns occupy a variable amount of space, but still at least N bytes.

3. For tables that are big, or contain lots of repetitive text or numeric data, consider using COMPRESSED row format. Less disk I/O is required to bring data into the buffer pool, or to perform full table scans. Before making a permanent decision, measure the amount of compression you can achieve by using COMPRESSED versus COMPACT row format. *Caveat:* Benchmarks rarely show better than 2:1 compression and there is a lot of overhead in the buffer_pool for COMPRESSED.
4. Once your data reaches a stable size, or a growing table has increased by tens or some hundreds of megabytes, consider using the OPTIMIZE TABLE statement to reorganize the table and compact any wasted space. The reorganized tables require less disk I/O to perform full table scans. This is a straightforward technique that can improve performance when other techniques such as improving index usage or tuning application code are not practical. *Caveat*: Regardless of table size, OPTIMIZE TABLE should only rarely be performed. This is because it is costly, and rarely improves the table enough to be worth it. InnoDB is reasonably good at keeping its B+Trees free of a lot of wasted space.

> OPTIMIZE TABLE copies the data part of the table and rebuilds the indexes. The benefits come from improved packing of data within indexes, and reduced fragmentation within the tablespaces and on disk. The benefits vary depending on the data in each table. You may find that there are significant gains for some and not for others, or that the gains decrease over time until you next optimize the table. This operation can be slow if the table is large or if the indexes being rebuilt do not fit into the buffer pool. The first run after adding a lot of data to a table is often much slower than later runs.

Chapter 69: Performance Tuning

Section 69.1: Don't hide in function

A common mistake is to hide an indexed column inside a function call. For example, this can't be helped by an index:

```
WHERE DATE(dt) = '2000-01-01'
```

Instead, given `INDEX(dt)` then these may use the index:

```
WHERE dt = '2000-01-01'  -- if `dt` is datatype `DATE`
```

This works for `DATE`, `DATETIME`, `TIMESTAMP`, and even `DATETIME(6)` (microseconds):

```
WHERE dt >= '2000-01-01'
  AND dt  < '2000-01-01' + INTERVAL 1 DAY
```

Section 69.2: OR

In general `OR` kills optimization.

```
WHERE a = 12 OR b = 78
```

cannot use `INDEX(a,b)`, and may or may not use `INDEX(a)`, `INDEX(b)` via "index merge". Index merge is better than nothing, but only barely.

```
WHERE x = 3 OR x = 5
```

is turned into

```
WHERE x IN (3, 5)
```

which *may* use an index with x in it.

Section 69.3: Add the correct index

This is a huge topic, but it is also the most important "performance" issue.

The main lesson for a novice is to learn of "composite" indexes. Here's a quick example:

```
INDEX(last_name, first_name)
```

is excellent for these:

```
WHERE last_name = '...'
WHERE first_name = '...' AND last_name = '...'    -- (order in WHERE does not matter)
```

but not for

```
WHERE first_name = '...'    -- order in INDEX _does_ matter
WHERE last_name = '...' OR first_name = '...'    -- "OR" is a killer
```

Section 69.4: Have an INDEX

The most important thing for speeding up a query on any non-tiny table is to have a suitable index.

```
WHERE a = 12  --> INDEX(a)
WHERE a > 12  --> INDEX(a)

WHERE a = 12 AND b > 78  --> INDEX(a,b) is more useful than INDEX(b,a)
WHERE a > 12 AND b > 78  --> INDEX(a) or INDEX(b); no way to handle both ranges

ORDER BY x  --> INDEX(x)
ORDER BY x, y  --> INDEX(x,y) in that order
ORDER BY x DESC, y ASC  --> No index helps - because of mixing ASC and DESC
```

Section 69.5: Subqueries

Subqueries come in several flavors, and they have different optimization potential. First, note that subqueries can be either "correlated" or "uncorrelated". Correlated means that they depend on some value from outside the subquery. This generally implies that the subquery *must* be re-evaluated for each outer value.

This correlated subquery is often pretty good. Note: It must return at most 1 value. It is often useful as an alternative to, though not necessarily faster than, a `LEFT JOIN`.

```
SELECT a, b, ( SELECT ... FROM t WHERE t.x = u.x ) AS c
    FROM u ...
SELECT a, b, ( SELECT MAX(x) ... ) AS c
    FROM u ...
SELECT a, b, ( SELECT x FROM t ORDER BY ... LIMIT 1 ) AS c
    FROM u ...
```

This is usually uncorrelated:

```
SELECT ...
    FROM ( SELECT ... ) AS a
    JOIN b ON ...
```

Notes on the `FROM-SELECT`:

- If it returns 1 row, great.
- A good paradigm (again "1 row") is for the subquery to be `( SELECT @n := 0 )`, thereby initializing an `@variable for use in the rest or the query.
- If it returns many rows *and* the `JOIN` also is `( SELECT ... )` with many rows, then efficiency can be terrible. Pre-5.6, there was no index, so it became a `CROSS JOIN`; 5.6+ involves deducing the best index on the temp tables and then generating it, only to throw it away when finished with the `SELECT`.

Section 69.6: JOIN + GROUP BY

A common problem that leads to an inefficient query goes something like this:

```
SELECT ...
    FROM a
    JOIN b  ON ...
    WHERE ...
    GROUP BY a.id
```

First, the `JOIN` expands the number of rows; then the `GROUP BY` whittles it back down the the number of rows in `a`.

There may not be any good choices to solve this explode-implode problem. One possible option is to turn the `JOIN` into a correlated subquery in the `SELECT`. This also eliminates the `GROUP BY`.

Section 69.7: Set the cache correctly

`innodb_buffer_pool_size` should be about 70% of available RAM.

Section 69.8: Negatives

Here are some things that are not likely to help performance. They stem from out-of-date information and/or naivety.

- InnoDB has improved to the point where MyISAM is unlikely to be better.
- `PARTITIONing` rarely provides performance benefits; it can even hurt performance.
- Setting `query_cache_size` bigger than 100M will usually *hurt* performance.
- Increasing lots of values in `my.cnf` may lead to 'swapping', which is a *serious* performance problem.
- "Prefix indexes" (such as `INDEX(foo(20))`) are generally useless.
- `OPTIMIZE TABLE` is almost always useless. (And it involves locking the table.)

Appendix A: Reserved Words

MySQL has some special names called *reserved words*. A reserved word can be used as an identifier for a table, column, etc. only if it's wrapped in backticks (`), otherwise it will give rise to an error.

To avoid such errors, either don't use reserved words as identifiers or wrap the offending identifier in backticks.

Section A.1: Errors due to reserved words

When trying to select from a table called `order` like this

```
select * from order
```

the error rises:

> Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'order' at line 1

Reserved keywords in MySQL need to be escaped with backticks (`)

```
select * from `order`
```

to distinguish between a keyword and a table or column name.

See also: Syntax error due to using a reserved word as a table or column name in MySQL.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

0x49D1	Chapter 10
4thfloorstudios	Chapter 12
a coder	Chapter 62
A J	Chapters 21, 24 and 30
Abubakkar	Chapter 20
Adam	Chapter 14
agold	Chapter 50
Alex Recarey	Chapter 31
alex9311	Chapter 25
Alvaro Flaño Larrondo	Chapter 6
Aman Dhanda	Chapter 1
Aminadav	Chapter 63
Andy	Chapter 1
andygeers	Chapter 25
Ani Menon	Chapters 3 and 53
animuson	Chapter 6
aries12	Chapter 52
arushi	Chapter 68
Aryo	Chapter 25
Asaph	Chapters 50 and 52
Asjad Athick	Chapter 3
Athafoud	Chapter 1
BacLuc	Chapter 67
Barranka	Chapters 19, 25, 31 and 50
Batsu	Chapters 11, 20, 50, 51, 54 and 63
Ben Visness	Chapter 31
Benvorth	Chapters 3, 16 and 25
Bhavin Solanki	Chapter 3
bhrached	Chapter 52
Blag	Chapter 37
CGritton	Chapter 10
ChintaMoney	Chapter 38
Chip	Chapter 3
Chris	Chapter 12
CodeWarrior	Chapters 1 and 38
CPHPython	Chapters 6 and 25
dakab	Chapter 2
Damian Yerrick	Chapter 15
Darwin von Corax	Chapters 25 and 30
Dinidu Hewage	Chapter 10
Dipen Shah	Chapter 1
Divya	Chapter 24
Drew	Chapters 2, 3, 4, 7, 10, 11, 12, 21, 25, 29, 30, 39 and 46
e4c5	Chapters 24, 34, 37 and 52
Eugene	Chapters 38, 56 and 62
falsefive	Chapter 60
Filipe Martins	Chapter 14

Florian Genser	Chapters 13 and 36
FMashiro	Chapters 17 and 61
ForguesR	Chapter 11
gabe3886	Chapters 10 and 11
Hardik Kanjariya ツ	Chapters 63 and 65
HCarrasko	Chapter 25
Horen	Chapter 10
Hugo Buff	Chapter 10
Ian Gregory	Chapter 6
Ian Kenney	Chapter 10
jan_kiran	Chapter 38
JayRizzo	Chapter 1
Jean Vitor	Chapter 25
John M	Chapter 25
JohnLBevan	Chapter 26
juergen d	Chapters 1, 2, 20, 29, 30 and 70
KalenGi	Chapter 50
KartikKannapur	Chapter 3
Khurram	Chapters 11 and 12
kolunar	Chapters 8, 20, 26, 30, 31 and 33
Kruti Patel	Chapter 3
Lahiru	Chapter 66
Lijo	Chapter 14
LiuYan 刘研	Chapter 26
llanato	Chapter 30
Lucas Paolillo	Chapter 29
Majid	Chapter 48
Marina K.	Chapter 24
Matas Vaitkevicius	Chapters 26, 36 and 37
Matt S	Chapters 6 and 21
Mattew Whitt	Chapter 18
Matthew	Chapter 50
matthiasunt	Chapter 16
mayojava	Chapter 26
Md. Nahiduzzaman Rose	Chapter 1
MohaMad	Chapters 2, 34 and 35
molavec	Chapter 41
Nate Vaughan	Chapter 57
Nathaniel Ford	Chapter 54
Nikita Kurtin	Chapter 24
Noah van der Aa	Chapter 27
O. Jones	Chapters 1, 3, 15, 19, 21, 22, 29, 32 and 59
Optimised Bikash Panda	Chapter 16
Parth Patel	Chapters 1 and 25
ParthaSen	Chapters 1, 11, 25 and 27
Philipp	Chapter 66
Ponnarasu	Chapters 10 and 16
R.K123	Chapters 3, 7, 11, 12, 24, 28, 29, 44, 45, 49, 50 and 55
RamenChef	Chapters 7, 24 and 27
ratchet	Chapter 50
RationalDev	Chapters 24 and 37
rene	Chapter 36
	Chapter 39

Richard Hamilton	Chapters 2 and 19
Rick	Chapters 16, 25 and 42
Rick James	Chapters 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 16, 18, 19, 20, 21, 24, 25, 26, 27, 28, 29, 30, 31, 33, 36, 38, 40, 42, 43, 44, 45, 46, 47, 48, 50, 52, 53, 54, 55, 58, 64, 65, 68 and 69
Riho	Chapters 10 and 18
Rodrigo Darti da Costa	Chapter 68
Romain Vincent	Chapter 1
Saroj Sasmal	Chapter 25
SeeuD1	Chapter 3
Sevle	Chapter 12
skytreader	Chapter 25
Stefan Rogin	Chapter 25
Steve Chambers	Chapter 26
still_learning	Chapters 7 and 50
strangeqargo	Chapters 10 and 11
Strawberry	Chapter 25
Sumit Gupta	Chapters 5 and 21
SuperDJ	Chapter 4
Tarik	Chapter 18
Thuta Aung	Chapters 14 and 26
Timothy	Chapter 25
Tushar patel	Chapter 27
user2314737	Chapters 23 and 70
user3617558	Chapter 16
user5389107	Chapters 1 and 10
user6655061	Chapter 8
userlond	Chapter 30
vijeeshin	Chapter 28
Viktor	Chapter 65
WAF	Chapters 9, 14, 20, 24, 26, 37, 50 and 63
wangengzheng	Chapter 18
Wenzhong	Chapter 56
whrrgarbl	Chapter 14
winter	Chapter 33
YCF_L	Chapters 1, 3, 9, 23, 25, 26, 27 and 37
ypercube□□	Chapter 11
yukoff	Chapter 44
Yury Fedorov	Chapter 25
zeppelin	Chapter 10

You may also like