

 **FREE eBook**

LEARNING MySQL

Free unaffiliated eBook created from
Stack Overflow contributors.

#mysql

Table of Contents

About.....	1
Chapter 1: Getting started with MySQL.....	2
Remarks.....	2
Versions.....	2
Examples.....	3
Getting Started.....	3
Information Schema Examples.....	7
Processlist.....	7
Stored Procedure Searching.....	7
Chapter 2: ALTER TABLE.....	8
Syntax.....	8
Remarks.....	8
Examples.....	9
Changing storage engine; rebuild table; change file_per_table.....	9
ALTER COLUMN OF TABLE.....	9
ALTER table add INDEX.....	9
Change auto-increment value.....	10
Changing the type of a primary key column.....	10
Change column definition.....	10
Renaming a MySQL database.....	10
Swapping the names of two MySQL databases.....	11
Renaming a MySQL table.....	12
Renaming a column in a MySQL table.....	12
Chapter 3: Arithmetic.....	14
Remarks.....	14
Examples.....	14
Arithmetic Operators.....	14
BIGINT.....	14
DOUBLE.....	15
Mathematical Constants.....	15

Pi.....	15
Trigonometry (SIN, COS).....	15
Sine.....	15
Cosine.....	15
Tangent.....	15
Arc Cosine (inverse cosine).....	16
Arc Sine (inverse sine).....	16
Arc Tangent (inverse tangent).....	16
Cotangent.....	16
Conversion.....	16
Rounding (ROUND, FLOOR, CEIL).....	17
Round a decimal number to an integer value.....	17
Round up a number.....	17
Round down a number.....	17
Round a decimal number to a specified number of decimal places.....	17
Raise a number to a power (POW).....	18
Square Root (SQRT).....	18
Random Numbers (RAND).....	18
Generate a random number.....	18
Random Number in a range.....	18
Absolute Value and Sign (ABS, SIGN).....	19
Chapter 4: Backticks.....	20
Examples.....	20
Backticks usage.....	20
Chapter 5: Backup using mysqldump.....	21
Syntax.....	21
Parameters.....	21
Remarks.....	22
Examples.....	22
Creating a backup of a database or table.....	22
Specifying username and password.....	23
Restoring a backup of a database or table.....	23

mysqldump from a remote server with compression.....	24
restore a gzipped mysqldump file without uncompressing.....	24
Backup direct to Amazon S3 with compression.....	24
Tranferring data from one MySQL server to another.....	24
Backup database with stored procedures and functions.....	25
Chapter 6: Change Password.....	26
Examples.....	26
Change MySQL root password in Linux.....	26
Change MySQL root password in Windows.....	27
Process.....	27
Chapter 7: Character Sets and Collations.....	28
Examples.....	28
Declaration.....	28
Connection.....	28
Which CHARACTER SET and COLLATION?.....	28
Setting character sets on tables and fields.....	29
Chapter 8: Clustering.....	30
Examples.....	30
Disambiguation.....	30
Chapter 9: Comment Mysql.....	31
Remarks.....	31
Examples.....	31
Adding comments.....	31
Commenting table definitions.....	31
Chapter 10: Configuration and tuning.....	33
Remarks.....	33
Examples.....	33
InnoDB performance.....	33
Parameter to allow huge data to insert.....	33
Increase the string limit for group_concat.....	34
Minimal InnoDB configuration.....	34
Secure MySQL encryption.....	35

Chapter 11: Connecting with UTF-8 Using Various Programming language.....	36
Examples.....	36
Python.....	36
PHP.....	36
Chapter 12: Converting from MyISAM to InnoDB.....	38
Examples.....	38
Basic conversion.....	38
Converting All Tables in one Database.....	38
Chapter 13: Create New User.....	39
Remarks.....	39
Examples.....	39
Create a MySQL User.....	39
Specify the password.....	39
Create new user and grant all privileges to schema.....	39
Renaming user.....	40
Chapter 14: Creating databases.....	41
Syntax.....	41
Parameters.....	41
Examples.....	41
Create database, users, and grants.....	41
MyDatabase.....	43
System Databases.....	43
Creating and Selecting a Database.....	44
Chapter 15: Customize PS1.....	45
Examples.....	45
Customize the MySQL PS1 with current database.....	45
Custom PS1 via MySQL configuration file.....	45
Chapter 16: Data Types.....	46
Examples.....	46
Implicit / automatic casting.....	46
VARCHAR(255) -- or not.....	46
INT as AUTO_INCREMENT.....	47

Others.....	47
Introduction (numeric).....	48
Integer Types.....	48
Fixed Point Types.....	48
Decimal.....	49
Floating Point Types.....	49
Bit Value Type.....	49
CHAR(n).....	50
DATE, DATETIME, TIMESTAMP, YEAR, and TIME.....	50
Chapter 17: Date and Time Operations.....	52
Examples.....	52
Now().....	52
Date arithmetic.....	52
Testing against a date range.....	53
SYSDATE(), NOW(), CURDATE().....	53
Extract Date from Given Date or DateTime Expression.....	53
Using an index for a date and time lookup.....	53
Chapter 18: Dealing with sparse or missing data.....	55
Examples.....	55
Working with columns containing NULL values.....	55
Chapter 19: DELETE.....	58
Syntax.....	58
Parameters.....	58
Examples.....	58
Delete with Where clause.....	58
Delete all rows from a table.....	58
LIMITing deletes.....	59
Multi-Table Deletes.....	59
foreign keys.....	60
Basic delete.....	61
DELETE vs TRUNCATE.....	61
Multi-table DELETE.....	61

Chapter 20: Drop Table	63
Syntax	63
Parameters	63
Examples	63
Drop Table	63
Drop tables from database	64
Chapter 21: Dynamic Un-Pivot Table using Prepared Statement	65
Examples	65
Un-pivot a dynamic set of columns based on condition	65
Chapter 22: ENUM	68
Examples	68
Why ENUM?	68
TINYINT as an alternative	68
VARCHAR as an alternative	69
Adding a new option	69
NULL vs NOT NULL	69
Chapter 23: Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause	71
Introduction	71
Remarks	71
Examples	72
Using and misusing GROUP BY	72
Misusing GROUP BY to return unpredictable results: Murphy's Law	72
Misusing GROUP BY with SELECT *, and how to fix it	73
ANY_VALUE()	74
Chapter 24: Error codes	75
Examples	75
Error code 1064: Syntax error	75
Error code 1175: Safe Update	75
Error code 1215: Cannot add foreign key constraint	76
1045 Access denied	77
1236 "impossible position" in Replication	77
2002, 2003 Cannot connect	78

1067, 1292, 1366, 1411 - Bad Value for number, date, default, etc.....	78
126, 127, 134, 144, 145.....	78
139.....	79
1366.....	79
126, 1054, 1146, 1062, 24.....	79
Chapter 25: Events.....	81
Examples.....	81
Create an Event.....	81
Schema for testing.....	81
Create 2 events, 1st runs daily, 2nd runs every 10 minutes.....	81
Show event statuses (different approaches).....	82
Random stuff to consider.....	83
Chapter 26: Extract values from JSON type.....	84
Introduction.....	84
Syntax.....	84
Parameters.....	84
Remarks.....	84
Examples.....	84
Read JSON Array value.....	84
JSON Extract Operators.....	85
Chapter 27: Full-Text search.....	87
Introduction.....	87
Remarks.....	87
Examples.....	87
Simple FULLTEXT search.....	87
Simple BOOLEAN search.....	87
Multi-column FULLTEXT search.....	88
Chapter 28: Group By.....	89
Syntax.....	89
Parameters.....	89
Remarks.....	89

Examples.....	89
GROUP BY USING SUM Function.....	89
Group By Using MIN function.....	90
GROUP BY USING COUNT Function.....	90
GROUP BY using HAVING.....	90
Group By using Group Concat.....	90
GROUP BY with AGGREGATE functions.....	91
Chapter 29: Handling Time Zones.....	94
Remarks.....	94
Examples.....	94
Retrieve the current date and time in a particular time zone.....	94
Convert a stored `DATE` or `DATETIME` value to another time zone.....	94
Retrieve stored `TIMESTAMP` values in a particular time zone.....	95
What is my server's local time zone setting?.....	95
What time_zone values are available in my server?.....	96
Chapter 30: Indexes and Keys.....	97
Syntax.....	97
Remarks.....	97
Concepts.....	97
Examples.....	98
Create index.....	98
Create unique index.....	98
Drop index.....	98
Create composite index.....	98
AUTO_INCREMENT key.....	98
Chapter 31: INSERT.....	100
Syntax.....	100
Remarks.....	100
Examples.....	101
Basic Insert.....	101
INSERT, ON DUPLICATE KEY UPDATE.....	101
Inserting multiple rows.....	101

Ignoring existing rows.....	102
INSERT SELECT (Inserting data from another Table).....	102
INSERT with AUTO_INCREMENT + LAST_INSERT_ID().....	103
Lost AUTO_INCREMENT ids.....	104
Chapter 32: Install Mysql container with Docker-Compose.....	106
Examples.....	106
Simple example with docker-compose.....	106
Chapter 33: Joins.....	107
Syntax.....	107
Examples.....	107
Joining Examples.....	107
JOIN with subquery ("Derived" table).....	107
Retrieve customers with orders -- variations on a theme.....	108
Full Outer Join.....	109
Inner-join for 3 tables.....	110
Joins visualized.....	111
Chapter 34: JOINS: Join 3 table with the same name of id.....	113
Examples.....	113
Join 3 tables on a column with the same name.....	113
Chapter 35: JSON.....	114
Introduction.....	114
Remarks.....	114
Examples.....	114
Create simple table with a primary key and JSON field.....	114
Insert a simple JSON.....	114
Insert mixed data into a JSON field.....	114
Updating a JSON field.....	115
CAST data to JSON type.....	115
Create Json Object and Array.....	115
Chapter 36: Limit and Offset.....	117
Syntax.....	117
Remarks.....	117

Examples.....	117
Limit and Offset relationship.....	117
LIMIT clause with one argument.....	117
LIMIT clause with two arguments.....	118
OFFSET keyword: alternative syntax.....	119
Chapter 37: LOAD DATA INFILE.....	120
Syntax.....	120
Examples.....	120
using LOAD DATA INFILE to load large amount of data to database.....	120
Import a CSV file into a MySQL table.....	121
Load data with duplicates.....	121
LOAD DATA LOCAL.....	121
LOAD DATA INFILE 'fname' REPLACE.....	121
LOAD DATA INFILE 'fname' IGNORE.....	122
Load via intermediary table.....	122
import / export.....	122
Chapter 38: Log files.....	123
Examples.....	123
A List.....	123
Slow Query Log.....	123
General Query Log.....	124
Error Log.....	126
Chapter 39: Many-to-many Mapping table.....	128
Remarks.....	128
Examples.....	128
Typical schema.....	128
Chapter 40: MyISAM Engine.....	129
Remarks.....	129
Examples.....	129
ENGINE=MyISAM.....	129
Chapter 41: MySQL Admin.....	130

Examples.....	130
Change root password.....	130
Drop database.....	130
Atomic RENAME & Table Reload.....	130
Chapter 42: MySQL client.....	131
Syntax.....	131
Parameters.....	131
Examples.....	131
Base login.....	131
Execute commands.....	132
Execute command from a string.....	132
Execute from script file:.....	133
Write the output on a file.....	133
Chapter 43: MySQL LOCK TABLE.....	134
Syntax.....	134
Remarks.....	134
Examples.....	134
Mysql Locks.....	134
Row Level Locking.....	135
Chapter 44: Mysql Performance Tips.....	138
Examples.....	138
Select Statement Optimization.....	138
Optimizing Storage Layout for InnoDB Tables.....	138
Building a composite index.....	139
Chapter 45: MySQL Unions.....	140
Syntax.....	140
Remarks.....	140
Examples.....	140
Union operator.....	140
Union ALL.....	141
UNION ALL With WHERE.....	141
Chapter 46: mysqlimport.....	143

Parameters.....	143
Remarks.....	143
Examples.....	143
Basic usage.....	143
Using a custom field-delimiter.....	144
Using a custom row-delimiter.....	144
Handling duplicate keys.....	144
Conditional import.....	145
Import a standard csv.....	145
Chapter 47: NULL.....	146
Examples.....	146
Uses for NULL.....	146
Testing NULLs.....	146
Chapter 48: One to Many.....	147
Introduction.....	147
Remarks.....	147
Examples.....	147
Example Company Tables.....	147
Get the Employees Managed by a Single Manager.....	148
Get the Manager for a Single Employee.....	148
Chapter 49: ORDER BY.....	149
Examples.....	149
Contexts.....	149
Basic.....	149
ASCending / DESCending.....	149
Some tricks.....	149
Chapter 50: Partitioning.....	151
Remarks.....	151
Examples.....	151
RANGE Partitioning.....	151
LIST Partitioning.....	152
HASH Partitioning.....	153

Chapter 51: Performance Tuning	154
Syntax	154
Remarks	154
Examples	154
Add the correct index	154
Set the cache correctly	155
Avoid inefficient constructs	155
Negatives	155
Have an INDEX	155
Don't hide in function	156
OR	156
Subqueries	156
JOIN + GROUP BY	157
Chapter 52: Pivot queries	158
Remarks	158
Examples	158
Creating a pivot query	158
Chapter 53: PREPARE Statements	160
Syntax	160
Examples	160
PREPARE, EXECUTE and DEALLOCATE PREPARE Statements	160
Construct and execute	160
Alter table with add column	161
Chapter 54: Recover and reset the default root password for MySQL 5.7+	162
Introduction	162
Remarks	162
Examples	162
What happens when the initial start up of the server	162
How to change the root password by using the default password	162
reset root password when " /var/run/mysqld" for UNIX socket file don't exists"	163
Chapter 55: Recover from lost root password	165
Examples	165

Set root password, enable root user for socket and http access.....	165
Chapter 56: Regular Expressions.....	166
Introduction.....	166
Examples.....	166
REGEXP / RLIKE.....	166
Pattern ^.....	166
Pattern \$**.....	166
NOT REGEXP.....	167
Regex Contain.....	167
Any character between [].....	167
Pattern or 	167
Counting regular expression matches.....	167
Chapter 57: Replication.....	169
Remarks.....	169
Examples.....	169
Master - Slave Replication Setup.....	169
Replication Errors.....	172
Chapter 58: Reserved Words.....	174
Introduction.....	174
Remarks.....	174
Examples.....	179
Errors due to reserved words.....	179
Chapter 59: Security via GRANTS.....	181
Examples.....	181
Best Practice.....	181
Host (of user@host).....	181
Chapter 60: SELECT.....	183
Introduction.....	183
Syntax.....	183
Remarks.....	183
Examples.....	183

SELECT by column name	183
SELECT all columns (*)	184
SELECT with WHERE	185
Query with a nested SELECT in the WHERE clause	185
SELECT with LIKE (%)	185
SELECT with Alias (AS)	187
SELECT with a LIMIT clause	187
SELECT with DISTINCT	188
SELECT with LIKE()	189
SELECT with CASE or IF	189
SELECT with BETWEEN	190
SELECT with date range	191
Chapter 61: Server Information	192
Parameters	192
Examples	192
SHOW VARIABLES example	192
SHOW STATUS example	192
Chapter 62: SSL Connection Setup	194
Examples	194
Setup for Debian-based systems	194
Generating a CA and SSL keys	194
Adding the keys to MySQL	194
Test the SSL connection	195
Enforcing SSL	195
References and further reading:	196
Setup for CentOS7 / RHEL7	196
First, log on to dbserver	196
END OF SERVER SIDE WORK FOR NOW	198
still on the client here	199
NOW WE ARE READY TO TEST THE SECURE CONNECTION	200
We're still on appclient here	200

Chapter 63: Stored routines (procedures and functions)	202
Parameters	202
Remarks	202
Examples	202
Create a Function	202
Create Procedure with a Constructed Prepare	203
Stored procedure with IN, OUT, INOUT parameters	204
Cursors	205
Multiple ResultSets	207
Create a function	207
Chapter 64: String operations	208
Parameters	208
Examples	210
Find element in comma separated list	210
STR_TO_DATE - Convert string to date	210
LOWER() / LCASE()	211
REPLACE()	211
SUBSTRING()	211
UPPER() / UCASE()	211
LENGTH()	212
CHAR_LENGTH()	212
HEX(str)	212
Chapter 65: Table Creation	213
Syntax	213
Remarks	213
Examples	213
Basic table creation	213
Setting defaults	214
Table creation with Primary Key	214
Defining one column as Primary Key (inline definition)	215
Defining a multiple-column Primary Key	215
Table creation with Foreign Key	216

Cloning an existing table.....	216
CREATE TABLE FROM SELECT.....	217
Show Table Structure.....	217
Table Create With TimeStamp Column To Show Last Update.....	218
Chapter 66: Temporary Tables.....	219
Examples.....	219
Create Temporary Table.....	219
Drop Temporary Table.....	219
Chapter 67: Time with subsecond precision.....	221
Remarks.....	221
Examples.....	221
Get the current time with millisecond precision.....	221
Get the current time in a form that looks like a Javascript timestamp.....	221
Create a table with columns to store sub-second time.....	222
Convert a millisecond-precision date / time value to text.....	222
Store a Javascript timestamp into a TIMESTAMP column.....	222
Chapter 68: Transaction.....	223
Examples.....	223
Start Transaction.....	223
COMMIT , ROLLBACK and AUTOCOMMIT.....	224
Transaction using JDBC Driver.....	226
Chapter 69: TRIGGERS.....	230
Syntax.....	230
Remarks.....	230
FOR EACH ROW.....	230
CREATE OR REPLACE TRIGGER.....	230
Examples.....	231
Basic Trigger.....	231
Types of triggers.....	231
Timing.....	231
Triggering event.....	232

Before Insert trigger example	232
Before Update trigger example	232
After Delete trigger example	232
Chapter 70: UNION	234
Syntax.....	234
Remarks.....	234
Examples.....	234
Combining SELECT statements with UNION.....	234
ORDER BY.....	234
Pagination via OFFSET.....	235
Combining data with different columns.....	235
UNION ALL and UNION.....	235
Combining and merging data on different MySQL tables with the same columns into unique row.....	236
Chapter 71: UPDATE	237
Syntax.....	237
Examples.....	237
Basic Update.....	237
Updating one row.....	237
Updating all rows.....	237
Update with Join Pattern.....	238
UPDATE with ORDER BY and LIMIT.....	238
Multiple Table UPDATE.....	239
Bulk UPDATE.....	239
Chapter 72: Using Variables	241
Examples.....	241
Setting Variables.....	241
Row Number and Group By using variables in Select Statement.....	242
Chapter 73: VIEW	244
Syntax.....	244
Parameters.....	244
Remarks.....	244

Examples.....	245
Create a View.....	245
A view from two tables.....	246
Updating a table via a VIEW.....	246
DROPPING A VIEW.....	246
Credits.....	248

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [mysql](#)

It is an unofficial and free MySQL ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official MySQL.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with MySQL

Remarks



[MySQL](#) is an open-source Relational Database Management System (RDBMS) that is developed and supported by Oracle Corporation.

MySQL is [supported](#) on a large number of platforms, including Linux variants, OS X, and Windows. It also has [APIs](#) for a large number of languages, including C, C++, Java, Lua, .Net, Perl, PHP, Python, and Ruby.

[MariaDB](#) is a fork of MySQL with a [slightly different feature set](#). It is entirely compatible with MySQL for most applications.

Versions

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

Version	Release Date
5.7	2015-10-01

Examples

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%';
```

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%';
```

Read [Getting started with MySQL](https://riptutorial.com/mysql/topic/302/getting-started-with-mysql) online: <https://riptutorial.com/mysql/topic/302/getting-started-with-mysql>

Chapter 2: ALTER TABLE

Syntax

- ALTER [IGNORE] TABLE tbl_name [**alter_specification** [, alter_specification] ...]
[partition_options]

Remarks

```
alter_specification: table_options
| ADD [COLUMN] col_name column_definition [FIRST | AFTER col_name ]
| ADD [COLUMN] (col_name column_definition,...)
| ADD {INDEX|KEY} [index_name] [index_type] (index_col_name,...) [index_option] ...
| ADD [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...) [index_option]
...
| ADD [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY] [index_name] [index_type]
(index_col_name,...) [index_option] ...
| ADD FULLTEXT [INDEX|KEY] [index_name] (index_col_name,...) [index_option] ...
| ADD SPATIAL [INDEX|KEY] [index_name] (index_col_name,...) [index_option] ...
| ADD [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (index_col_name,...)
reference_definition
| ALGORITHM [=] {DEFAULT|INPLACE|COPY}
| ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
| CHANGE [COLUMN] old_col_name new_col_name column_definition [FIRST|AFTER col_name]
| LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}
| MODIFY [COLUMN] col_name column_definition [FIRST | AFTER col_name]
| DROP [COLUMN] col_name
| DROP PRIMARY KEY
| DROP {INDEX|KEY} index_name
| DROP FOREIGN KEY fk_symbol
| DISABLE KEYS
| ENABLE KEYS
| RENAME [TO|AS] new_tbl_name
| RENAME {INDEX|KEY} old_index_name TO new_index_name
| ORDER BY col_name [, col_name] ...
| CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]
| [DEFAULT] CHARACTER SET [=] charset_name [COLLATE [=] collation_name]
| DISCARD TABLESPACE
| IMPORT TABLESPACE
| FORCE
| {WITHOUT|WITH} VALIDATION
| ADD PARTITION (partition_definition)
| DROP PARTITION partition_names
| DISCARD PARTITION {partition_names | ALL} TABLESPACE
| IMPORT PARTITION {partition_names | ALL} TABLESPACE
| TRUNCATE PARTITION {partition_names | ALL}
| COALESCE PARTITION number
| REORGANIZE PARTITION partition_names INTO (partition_definitions)
| EXCHANGE PARTITION partition_name WITH TABLE tbl_name [{WITH|WITHOUT} VALIDATION]
| ANALYZE PARTITION {partition_names | ALL}
| CHECK PARTITION {partition_names | ALL}
| OPTIMIZE PARTITION {partition_names | ALL}
| REBUILD PARTITION {partition_names | ALL}
| REPAIR PARTITION {partition_names | ALL}
| REMOVE PARTITIONING
```

```

    | UPGRADE PARTITIONING
index_col_name: col_name [(length)] [ASC | DESC]
index_type: USING {BTREE | HASH}
index_option: KEY_BLOCK_SIZE [=] value
              | index_type
              | WITH PARSEr parser_name
              | COMMENT 'string'

```

table_options: table_option [[,] table_option] ... (see [CREATE TABLE](#) options)

partition_options: (see [CREATE TABLE](#) options)

Ref: [MySQL 5.7 Reference Manual / ... / ALTER TABLE Syntax / 14.1.8 ALTER TABLE Syntax](#)

Examples

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`.

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

```

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`)
```

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;
```

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.

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
```

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.

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

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).

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.*

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.

Read ALTER TABLE online: <https://riptutorial.com/mysql/topic/2627/alter-table>

Chapter 3: Arithmetic

Remarks

MySQL, on most machines, uses [64-bit IEEE 754 floating point arithmetic](#) for its calculations.

In integer contexts it uses integer arithmetic.

- `RAND()` is not a perfect random number generator. It is mainly used to quickly generate pseudorandom numbers

Examples

Arithmetic Operators

MySQL provides the following arithmetic operators

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

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](#).

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
```

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 \$\epsilon\$ \(epsilon\) 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: it 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
```

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.

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
```

Square Root (SQRT)

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

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

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 $a \leq n \leq b$, 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.

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	$n < 0$	<code>SELECT SIGN(42); -> 1</code>
0	$n = 0$	<code>SELECT SIGN(0); -> 0</code>
1	$n > 0$	<code>SELECT SIGN(-3); -> -1</code>

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

Read Arithmetic online: <https://riptutorial.com/mysql/topic/4516/arithmetic>

Chapter 4: Backticks

Examples

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 queries 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.

Read Backticks online: <https://riptutorial.com/mysql/topic/5208/backticks>

Chapter 5: Backup using mysqldump

Syntax

- `mysqldump -u [username] -p[password] [other options] db_name > dumpFileName.sql ///` To Backup single database
- `mysqldump -u [username] -p[password] [other options] db_name [tbl_name1 tbl_name2 tbl_name2 ...] > dumpFileName.sql ///` To Backup one or more tables
- `mysqldump -u [username] -p[password] [other options] --databases db_name1 db_name2 db_name3 ... > dumpFileName.sql ///` To Backup one or more complete databases
- `mysqldump -u [username] -p[password] [other options] --all-databases > dumpFileName.sql ///` To Backup entire MySQL server

Parameters

Option	Effect
--	# Server login options
-h (--host)	Host (IP address or hostname) to connect to. Default is <code>localhost</code> (127.0.0.1) Example: <code>-h localhost</code>
-u (--user)	MySQL user
-p (--password)	MySQL password. Important: When using <code>-p</code> , there must not be a space between the option and the password. Example: <code>-pMyPassword</code>
--	# Dump options
--add-drop-database	Add a <code>DROP DATABASE</code> statement before each <code>CREATE DATABASE</code> statement. Useful if you want to replace databases in the server.
--add-drop-table	Add a <code>DROP TABLE</code> statement before each <code>CREATE TABLE</code> statement. Useful if you want to replace tables in the server.
--no-create-db	Suppress the <code>CREATE DATABASE</code> 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 <code>CREATE TABLE</code> 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 <code>CREATE TABLE</code> statements. Useful for creating "template" databases

Option	Effect
<code>-R (--routines)</code>	Include stored procedures / functions in the dump.
<code>-K (--disable-keys)</code>	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.

Remarks

The output of a `mysqldump` operation is a lightly commented file containing sequential SQL statements that are compatible with the version of MySQL utilities that was used to generate it (with attention paid to compatibility with previous versions, but no guarantee for future ones). Thus, the restoration of a `mysqldumped` database comprises execution of those statements. Generally, this file

- **DROPS** the first specified table or view
- **CREATEs** that table or view
- For tables dumped with data (i.e. without the `--no-data` option)
 - **LOCKs** the table
 - **INSERTs** all of the rows from the original table in one statement
- **UNLOCK TABLES**
- Repeats the above for all other tables and views
- **DROPS** the first included routine
- **CREATEs** that routine
- Repeats the same for all other routines

The presence of the **DROP** before **CREATE** for each table means that if the schema is present, whether or not it is empty, using a `mysqldump` file for its restoration will populate or overwrite the data therein.

Examples

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.

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 your 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]
```

(Explicitly specifying the password on the commandline is Not Recommended due to security concerns.)

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
```

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
```

restore a gzipped mysqldump file without uncompressing

```
gunzip -c dbname.sql.gz | mysql dbname -u username -p
```

Note: `-c` means write output to stdout.

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.

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.

Backup database with stored procedures and functions

By default stored procedures and functions are 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`.

Read Backup using `mysqldump` online: <https://riptutorial.com/mysql/topic/604/backup-using-mysqldump>

Chapter 6: Change Password

Examples

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.

5.7

```
FLUSH PRIVILEGES;  
ALTER USER 'root'@'localhost' IDENTIFIED BY 'new_password';  
FLUSH PRIVILEGES;  
exit;
```

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
```

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 :

Press **Ctrl+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');
```

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>

Read Change Password online: <https://riptutorial.com/mysql/topic/2761/change-password>

Chapter 7: Character Sets and Collations

Examples

Declaration

```
CREATE TABLE foo ( ...  
    name CHARACTER SET utf8mb4  
    ... );
```

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`.

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.

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.

Read Character Sets and Collations online: <https://riptutorial.com/mysql/topic/4569/character-sets-and-collations>

Chapter 8: Clustering

Examples

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`.

Read Clustering online: <https://riptutorial.com/mysql/topic/5130/clustering>

Chapter 9: Comment Mysql

Remarks

The `--` style of comment, which requires a trailing space, [differs in behavior from the SQL standard](#), which does not require the space.

Examples

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.
```

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`.

Read Comment Mysql online: <https://riptutorial.com/mysql/topic/2337/comment-mysql>

Chapter 10: Configuration and tuning

Remarks

Configuration happens in one of 3 ways:

- command line options
- the `my.cnf` configuration file
- setting variables from within the server

Command Line options takes the form `mysqld --long-parameter-name=value --another-parameter`. The same parameters can be placed in the `my.cnf` configuration file. *Some* parameters are configurable using system variables from within MySQL. Check the official documentation for a complete list of parameters.

Variables can have dash `-` or underscore `_`. Spaces may exist around the `=`. Large numbers can be suffixed by `K`, `M`, `G` for kilo-, mega-, and giga-. One setting per line.

Flags: Usually `ON` and `1` are synonymous, ditto for `OFF` and `0`. Some flags have nothing after them.

When placing the settings in `my.cnf`, all settings for the *server* must be in the `[mysqld]` section, so don't blindly add settings to the end of the file. (Note: For tools that allow multiple mysql instances to share one `my.cnf`, the section names may be different.)

Examples

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.

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

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.

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 `DROPEd`. 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
```

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
```

Read Configuration and tuning online: <https://riptutorial.com/mysql/topic/3134/configuration-and-tuning>

Chapter 11: Connecting with UTF-8 Using Various Programming language.

Examples

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" />
```

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);
```

Read Connecting with UTF-8 Using Various Programming language. online:

<https://riptutorial.com/mysql/topic/7332/connecting-with-utf-8-using-various-programming-language->

Chapter 12: Converting from MyISAM to InnoDB

Examples

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](#)

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.

Read [Converting from MyISAM to InnoDB online](#):

<https://riptutorial.com/mysql/topic/3135/converting-from-myisam-to-innodb>

Chapter 13: Create New User

Remarks

To view a List of MySQL Users, we use the following command :

```
SELECT User,Host FROM mysql.user;
```

Examples

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';
```

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';
```

Create new user and grant all privileges 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

Renaming user

```
rename user 'user'@'%' to 'new_name'@'%';
```

If you create a user by mistake, you can change his name

Read Create New User online: <https://riptutorial.com/mysql/topic/3508/create-new-user>

Chapter 14: Creating databases

Syntax

- `CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name [create_specification] ///` To create database
- `DROP {DATABASE | SCHEMA} [IF EXISTS] db_name ///` To drop database

Parameters

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

Examples

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.

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`.

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)

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: *****
```

Read Creating databases online: <https://riptutorial.com/mysql/topic/600/creating-databases>

Chapter 15: Customize PS1

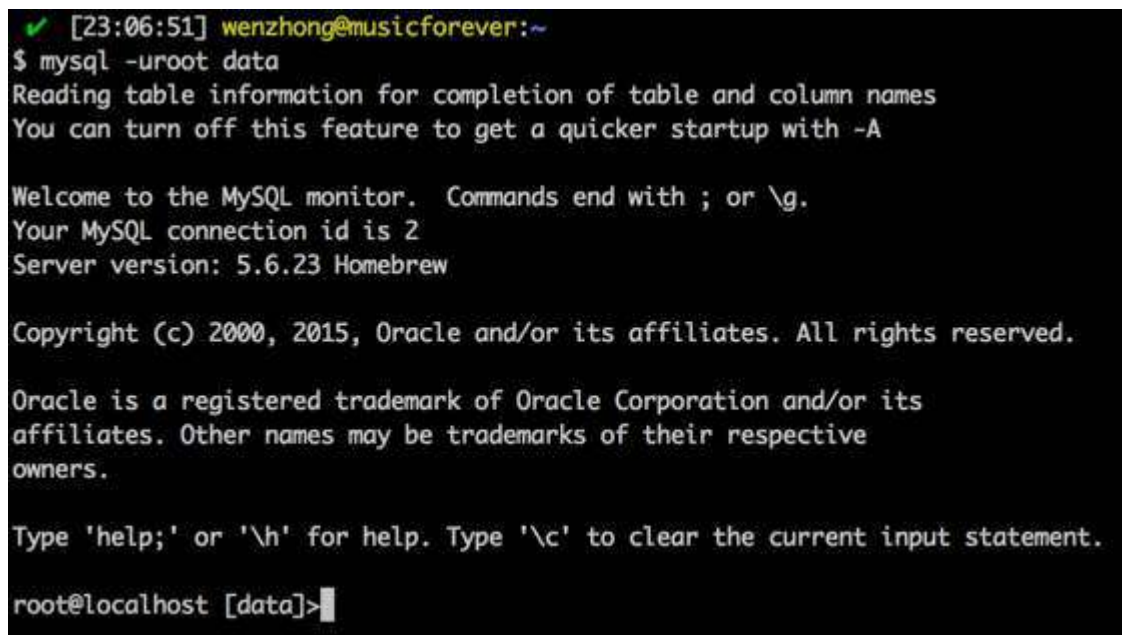
Examples

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].



```
✓ [23:06:51] wenzhong@musicforever:~  
$ mysql -uroot data  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 2  
Server version: 5.6.23 Homebrew  
  
Copyright (c) 2000, 2015, 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.  
root@localhost [data]>
```

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.

Read Customize PS1 online: <https://riptutorial.com/mysql/topic/5795/customize-ps1>

Chapter 16: Data Types

Examples

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

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 `VARCHARs` 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 `SELECTs` due to how temp tables are handled.

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.

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-destruct when my suggestions have been satisfied or vetoed.)

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

Integer Types

Minimal unsigned value is always 0.

Type	Storage (Bytes)	Minimum Value (Signed)	Maximum Value (Signed)	Maximum Value (Unsigned)
TINYINT	1	-2^7 -128	2^7-1 127	2^8-1 255
SMALLINT	2	-2^{15} -32,768	$2^{15}-1$ 32,767	$2^{16}-1$ 65,535
MEDIUMINT	3	-2^{23} -8,388,608	$2^{23}-1$ 8,388,607	$2^{24}-1$ 16,777,215
INT	4	-2^{31} -2,147,483,648	$2^{31}-1$ 2,147,483,647	$2^{32}-1$ 4,294,967,295
BIGINT	8	-2^{63} -9,223,372,036,854,775,808	$2^{63}-1$ 9,223,372,036,854,775,807	$2^{64}-1$ 18,446,744,073,709,551,615

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* $M/2$.

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.

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 $(1 \ll 7)$ for 128.

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

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 $4 \times n$ *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
```

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](#).

Read Data Types online: <https://riptutorial.com/mysql/topic/4137/data-types>

Chapter 17: Date and Time Operations

Examples

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'
```

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.

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`.

`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.

Extract Date from Given Date or DateTime Expression

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

The output will be:

```
2003-12-31
```

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.

Read Date and Time Operations online: <https://riptutorial.com/mysql/topic/1882/date-and-time-operations>

Chapter 18: Dealing with sparse or missing data

Examples

Working with columns containing 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
1	Initech	2013-01-31
2	Woodworking.com	2016-08-25
2	NY Times	2013-11-10
3	NFL.com	2014-04-13

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.

Read Dealing with sparse or missing data online: <https://riptutorial.com/mysql/topic/5866/dealing-with-sparse-or-missing-data>

Chapter 19: DELETE

Syntax

- `DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM table [WHERE conditions] [ORDER BY expression [ASC | DESC]] [LIMIT number_rows];` /// Syntax for delete row(s) from single table

Parameters

Parameter	Details
LOW_PRIORITY	If <code>LOW_PRIORITY</code> is provided, the delete will be delayed until there are no processes reading from the table
IGNORE	If <code>IGNORE</code> 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 <code>ORDER BY</code> is provided, records will be deleted in the given order
LIMIT	It controls the maximum number of records to delete from the table. Given <code>number_rows</code> will be deleted.

Examples

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.

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.

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.

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 involves tables with a foreign key constraint 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 foreign 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;
```

Basic delete

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

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

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 were overwritten. (The space cannot immediately be recouped for

`innodb_file_per_table=OFF`.)

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'
```

Read DELETE online: <https://riptutorial.com/mysql/topic/1487/delete>

Chapter 20: Drop Table

Syntax

- DROP TABLE table_name;
- DROP TABLE IF EXISTS table_name; -- to avoid pesky error in automated script
- DROP TABLE t1, t2, t3; -- DROP multiple tables
- DROP TEMPORARY TABLE t; -- DROP a table from CREATE TEMPORARY TABLE ...

Parameters

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.

Examples

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.

Drop tables from database

DROP TABLE Database.table_name

Read Drop Table online: <https://riptutorial.com/mysql/topic/4123/drop-table>

Chapter 21: Dynamic Un-Pivot Table using Prepared Statement

Examples

Un-pivot a dynamic set of columns based on condition

The following example is a very useful basis when you are trying to convert transaction data to un-pivoted data for BI/reporting reasons, where the dimensions which are to be un-pivoted can have a dynamic set of columns.

For our example, we suppose that the raw data table contains employee assessment data in the form of marked questions.

The raw data table is the following:

```
create table rawdata
(
  PersonId VARCHAR(255)
,Question1Id INT(11)
,Question2Id INT(11)
,Question3Id INT(11)
)
```

The rawdata table is a temporary table as part of the ETL procedure and can have a varying number of questions. The goal is to use the same un-pivoting procedure for an arbitrary number of Questions, namely columns that are going to be un-pivoted.

Below is a toy example of rawdata table:

	PersonId	Question1Id	Question2Id	Question3Id
	Giannaros	1	3	1
	Patra	2	4	3

The well-known,static way to unpivot the data, in MYSQL is by using UNION ALL:

```
create table unpivoteddata
(
  PersonId VARCHAR(255)
,QuestionId VARCHAR(255)
,QuestionValue INT(11)
);

INSERT INTO unpivoteddata SELECT PersonId, 'Question1Id' col, Question1Id
FROM rawdata
```

```

UNION ALL
SELECT PersonId, 'Question2Id' col, Question2Id
FROM rawdata
UNION ALL
SELECT PersonId, 'Question3Id' col, Question3Id
FROM rawdata;

```

In our case we want to define a way to unpivot an arbitrary number of QuestionId columns. For that we need to execute a prepared statement that is a dynamic select of the desired columns. In order to be able to choose which columns need to be un-pivoted, we will use a GROUP_CONCAT statement and we will choose the columns for which the data type is set to 'int'. In the GROUP_CONCAT we also include all additional elements of our SELECT statement to-be executed.

```

set @temp2 = null;

SELECT GROUP_CONCAT(' SELECT ', 'PersonId', ',', '', '', COLUMN_NAME, '', ' col
', ',', '', COLUMN_NAME, ' FROM rawdata' separator ' UNION ALL' ) FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'rawdata' AND DATA_TYPE = 'Int' INTO @temp2;

select @temp2;

```

In another occasion we could have chosen columns that the column name matches a pattern, for example instead of

```
DATA_TYPE = 'Int'
```

use

```
COLUMN_NAME LIKE 'Question%'
```

or something suitable that can be controlled through the ETL phase.

The prepared statement is finalized as follows:

```

set @temp3 = null;

select concat('INSERT INTO unpivoteddata', @temp2) INTO @temp3;

select @temp3;

prepare stmt FROM @temp3;
execute stmt;
deallocate prepare stmt;

```

The unpivoteddata table is the following:

```
SELECT * FROM unpivoteddata
```

PersonId	QuestionId	QuestionValue
Giannaros	Question1Id	1
Patra	Question1Id	2
Giannaros	Question2Id	3
Patra	Question2Id	4
Giannaros	Question3Id	1
Patra	Question3Id	3

Selecting columns according to a condition and then crafting a prepared statement is an efficient way of dynamically un-pivoting data.

Read [Dynamic Un-Pivot Table using Prepared Statement](https://riptutorial.com/mysql/topic/6491/dynamic-un-pivot-table-using-prepared-statement) online:

<https://riptutorial.com/mysql/topic/6491/dynamic-un-pivot-table-using-prepared-statement>

Chapter 22: ENUM

Examples

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
```

TINYINT as an alternative

Let's say we have

```
type ENUM('fish', 'mammal', 'bird')
```

An alternative is

```
type TINYINT UNSIGNED
```

plus

```
CREATE TABLE AnimalTypes (
  type TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(20) NOT NULL COMMENT "('fish','mammal','bird')",
  PRIMARY KEY(type),
  INDEX(name)
) ENGINE=InnoDB
```

which is very much like a many-to-many table.

Comparison, and whether better or worse than ENUM:

- (worse) INSERT: need to lookup the `type`
- (worse) SELECT: need to JOIN to get the string (ENUM gives you the string with no effort)
- (better) Adding new types: Simply insert into this table. With ENUM, you need to do an ALTER TABLE.
- (same) Either technique (for up to 255 values) takes only 1 byte.
- (mixed) There's also an issue of data integrity: `TINYINT` will admit invalid values; whereas `ENUM`

sets them to a special empty-string value (unless strict SQL mode is enabled, in which case they are rejected). Better data integrity can be achieved with `TINYINT` by making it a foreign key into a lookup table: which, with appropriate queries/joins, but there is still the small cost of reaching into the other table. (`FOREIGN KEYS` are not free.)

VARCHAR as an alternative

Let's say we have

```
type ENUM('fish','mammal','bird')
```

An alternative is

```
type VARCHAR(20) COMMENT "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

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.)

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)

```

Read ENUM online: <https://riptutorial.com/mysql/topic/4425/enum>

Chapter 23: Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause ...

Introduction

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.

Remarks

For a long time now, MySQL has contained a notorious nonstandard extension to `GROUP BY`, which allows oddball behavior in the name of efficiency. This extension has allowed countless developers around the world to use `GROUP BY` in production code without completely understanding what they were doing.

In particular, it's a bad idea to use `SELECT *` in a `GROUP BY` query, because a standard `GROUP BY` clause requires enumerating the columns. Many developers have, unfortunately, done that.

Read this. <https://dev.mysql.com/doc/refman/5.7/en/group-by-handling.html>

The MySQL team has been trying to fix this misfeature without messing up production code. They added a `sql_mode` flag in 5.7.5 named `ONLY_FULL_GROUP_BY` to compel standard behavior. In a recent release, they turned on that flag by default. When you upgraded your local MySQL to 5.7.14, the flag got switched on and your production code, dependent on the old extension, stopped working.

If you've recently started getting 1055 errors, what are your choices?

1. fix the offending SQL queries, or get their authors to do that.
2. roll back to a version of MySQL compatible out-of-the-box with the application software you use.
3. change your server's `sql_mode` to get rid of the newly set `ONLY_FULL_GROUP_BY` mode.

You can change the mode by doing a `SET` command.

```
SET sql_mode =  
'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_EN
```

should do the trick if you do it right after your application connects to MySQL.

Or, you can find [the init file in your MySQL installation](#), locate the `sql_mode=` line, and change it to

omit `ONLY_FULL_GROUP_BY`, and restart your server.

Examples

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`.

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.

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.

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.

Read [Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause ... online](https://riptutorial.com/mysql/topic/8245/error-1055--only-full-group-by--something--is-not-in-group-by-clause----):
<https://riptutorial.com/mysql/topic/8245/error-1055--only-full-group-by--something--is-not-in-group-by-clause---->

Chapter 24: Error codes

Examples

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.

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;
```

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.

1045 Access denied

See discussions in "GRANT" and "Recovering root password".

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, ...?)

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.

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?

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
```

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

1366

This usually means that the character set handling was not consistent between client and server. See ... for further assistance.

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 occurs 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 your 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 assign 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 discards it without generating any error.

```
INSERT IGNORE INTO userDetails VALUES (NULL , 'John', 'Doe', 1);
```

Read Error codes online: <https://riptutorial.com/mysql/topic/895/error-codes>

Chapter 25: Events

Examples

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 flexibility 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](#).

Read Events online: <https://riptutorial.com/mysql/topic/4319/events>

Chapter 26: Extract values from JSON type

Introduction

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.

Syntax

- `JSON_EXTRACT(json_doc,path[,...])`
- `JSON_EXTRACT(json_doc,path)`
- `JSON_EXTRACT(json_doc,path1,path2)`

Parameters

Parameter	Description
json_doc	valid JSON document
path	members path

Remarks

Mentioned in [MySQL 5.7 Reference Manual](#)

- Multiple matched values by path argument(s)

If it is possible that those arguments could return multiple values, the matched values are autowrapped as an array, in the order corresponding to the paths that produced them. Otherwise, the return value is the single matched value.

- `NULL` Result when:
 - any argument is `NULL`
 - path not matched

Returns `NULL` if any argument is `NULL` or no paths locate a value in the document.

Examples

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"]
```

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 ;
-- visuell:
  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\(+\)](#)

Read Extract values from JSON type online: <https://riptutorial.com/mysql/topic/9042/extract-values-from-json-type>

Chapter 27: Full-Text search

Introduction

MySQL offers FULLTEXT searching. It searches tables with columns containing text for the best matches for words and phrases.

Remarks

FULLTEXT searching works strangely on tables containing small numbers of rows. So, when you're experimenting with it, you may find it helpful to obtain a medium-sized table online. Here's a [table of book items](#), with titles and authors. You can download it, unzip it, and load it into MySQL.

FULLTEXT search is intended for use with human assistance. It's designed to yield more matches than an ordinary `WHERE column LIKE 'text%'` filtering operation.

FULLTEXT search is available for `MyISAM` tables. It is also available for `InnoDB` tables in MySQL version 5.6.4 or later.

Examples

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);
```

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);
```

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);
```

Read Full-Text search online: <https://riptutorial.com/mysql/topic/8759/full-text-search>

Chapter 28: Group By

Syntax

1. SELECT expression1, expression2, ... expression_n,
2. aggregate_function (expression)
3. FROM tables
4. [WHERE conditions]
5. GROUP BY expression1, expression2, ... expression_n;

Parameters

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	The 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.

Remarks

The MySQL GROUP BY clause is used in a SELECT statement to collect data across multiple records and group the results by one or more columns.

Its behavior is governed in part by the value of [the ONLY_FULL_GROUP_BY variable](#). When this is enabled, SELECT statements that group by any column not in the output return an error. ([This is the default as of 5.7.5.](#)) Both setting and not setting this variable can cause problems for naive users or users accustomed to other DBMSs.

Examples

GROUP BY USING SUM Function

```
SELECT product, SUM(quantity) AS "Total quantity"
FROM order_details
GROUP BY product;
```

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".

GROUP BY USING COUNT Function

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

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".

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-

Name	Score
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- |
+-----+-----+
```

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      |
+-----+-----+
```

Read Group By online: <https://riptutorial.com/mysql/topic/3523/group-by>

Chapter 29: Handling Time Zones

Remarks

When you need to handle time information for a worldwide user base in MySQL, use the `TIMESTAMP` data type in your tables.

For each user, store a user-preference timezone column. `VARCHAR(64)` is a good data type for that column. When a user registers to use your system, ask for the time zone value. Mine is Atlantic Time, `America/Edmonton`. Yours might or might not be `Asia/Kolkata` or `Australia/NSW`. For a user interface for this user-preference setting, the WordPress.org software has a good example.

Finally, whenever you establish a connection from your host program (Java, php, whatever) to your DBMS on behalf of a user, issue the SQL command

```
SET SESSION time_zone='(whatever tz string the user gave you)'
```

before you handle any user data involving times. Then all the `TIMESTAMP` times you have install will render in the user's local time.

This will cause all times going in to your tables to be converted to UTC, and all times coming out to be translated to local. It works properly for `NOW()` and `CURDATE()`. Again, you must use `TIMESTAMP` and not `DATETIME` or `DATE` data types for this.

Make sure your server OS and default MySQL time zones are set to UTC. If you don't do this before you start loading information into your database, it will be almost impossible to fix. If you use a vendor to run MySQL, insist they get this right.

Examples

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();
```

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'
```

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.

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, TIMESTAMPTDIFF(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 `TIMESTAMPS` 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. `TIMESTAMPS` are always translated from their stored UTC form to local time in `SELECT` statements. `DATETIMES` are not. So the [TIMESTAMPTDIFF\(MINUTE...\) operation](#) computes the difference between local and universal time.

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 approximately 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.

Read Handling Time Zones online: <https://riptutorial.com/mysql/topic/7849/handling-time-zones>

Chapter 30: Indexes and Keys

Syntax

- -- Create simple index

```
CREATE INDEX index_name ON table_name(column_name1 [, column_name2, ...])
```

- -- Create unique index

```
CREATE UNIQUE INDEX index_name ON table_name(column_name1 [, column_name2, ...])
```

- -- Drop index

```
DROP INDEX index_name ON tbl_name [algorithm_option | lock_option] ...
```

algorithm_option: ALGORITHM [=] {DEFAULT|INPLACE|COPY}

lock_option: LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}

Remarks

Concepts

An index in a MySQL table works like an index in a book.

Let's say you have a book about databases and you want to find some information about, say, storage. Without an index (assuming no other aid, such as a table of contents) you'd have to go through the pages one by one, until you found the topic (that's a "full table scan"). On the other hand, an index has a list of keywords, so you'd consult the index and see that storage is mentioned on pages 113-120, 231, and 354. Then you could flip to those pages directly, without searching (that's a search with an index, somewhat faster).

Of course, the usefulness of the index depends on many things - a few examples, using the simile above:

- If you had a book on databases and indexed the word "database", you might see that it's mentioned on pages 1-59, 61-290, and 292-400. That's a lot of pages, and in such a case, the index is not much help and it might be faster to go through the pages one by one. (In a database, this is "poor selectivity".)
- For a 10-page book, it makes no sense to make an index, as you may end up with a 10-page book prefixed by a 5-page index, which is just silly - just scan the 10 pages and be done with it.
- The index also needs to be useful - there's generally no point to indexing, for example, the

frequency of the letter "L" per page.

Examples

Create index

```
-- Create an index for column 'name' in table 'my_table'  
CREATE INDEX idx_name ON my_table(name);
```

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);
```

Drop index

```
-- Drop an index for column 'name' in table 'my_table'  
DROP INDEX idx_name ON my_table;
```

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 usually 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.

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.

Read Indexes and Keys online: <https://riptutorial.com/mysql/topic/1748/indexes-and-keys>

Chapter 31: INSERT

Syntax

1. `INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE] [INTO] tbl_name [PARTITION (partition_name,...)] [(col_name,...)] {VALUES | VALUE} ({expr | DEFAULT},...),(...),... [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]`
2. `INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE] [INTO] tbl_name [PARTITION (partition_name,...)] SET col_name={expr | DEFAULT}, ... [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]`
3. `INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO] tbl_name [PARTITION (partition_name,...)] [(col_name,...)] SELECT ... [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]`
4. An expression `expr` can refer to any column that was set earlier in a value list. For example, you can do this because the value for `col2` refers to `col1`, which has previously been assigned:
`INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);`
5. `INSERT` statements that use `VALUES` syntax can insert multiple rows. To do this, include multiple lists of column values, each enclosed within parentheses and separated by commas. Example:
`INSERT INTO tbl_name (a,b,c) VALUES(1,2,3),(4,5,6),(7,8,9);`
6. The values list for each row must be enclosed within parentheses. The following statement is illegal because the number of values in the list does not match the number of column names:
`INSERT INTO tbl_name (a,b,c) VALUES(1,2,3,4,5,6,7,8,9);`
7. **INSERT ... SELECT Syntax**
`INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO] tbl_name [PARTITION (partition_name,...)] [(col_name,...)] SELECT ... [ON DUPLICATE KEY UPDATE col_name=expr, ...]`
8. With `INSERT ... SELECT`, you can quickly insert many rows into a table from one or many tables. For example:
`INSERT INTO tbl_temp2 (fld_id) SELECT tbl_temp1.fld_order_id FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;`

Remarks

[Official INSERT Syntax](#)

Examples

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

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).

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');
```

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.

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 will 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
+----+-----+-----+
```

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 `DELETEs` 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.

Read `INSERT` online: <https://riptutorial.com/mysql/topic/866/insert>

Chapter 32: Install Mysql container with Docker-Compose

Examples

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
```

Read Install Mysql container with Docker-Compose online:

<https://riptutorial.com/mysql/topic/4458/install-mysql-container-with-docker-compose>

Chapter 33: Joins

Syntax

- `INNER` and `OUTER` are ignored.
- `FULL` is not implemented in MySQL.
- "commajoin" (`FROM a,b WHERE a.x=b.y`) is frowned on; use `FROM a JOIN b ON a.x=b.y` instead.
- `FROM a JOIN b ON a.x=b.y` includes rows that match in both tables.
- `FROM a LEFT JOIN b ON a.x=b.y` includes all rows from `a`, plus matching data from `b`, or `NULLs` if there is no matching row.

Examples

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;
```

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`.)

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;

```

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)
```

Inner-join for 3 tables

let's assume we have three table which can be used for simple website with Tags.

- First table is for Posts.

- Second for Tags
- Third for Tags & Post relation

first table "videogame"

id	title	reg_date	Content
1	BioShock Infinite	2016-08-08

"tags" table

id	name
1	yennefer
2	elizabeth

"tags_meta" table

post_id	tag_id
1	2

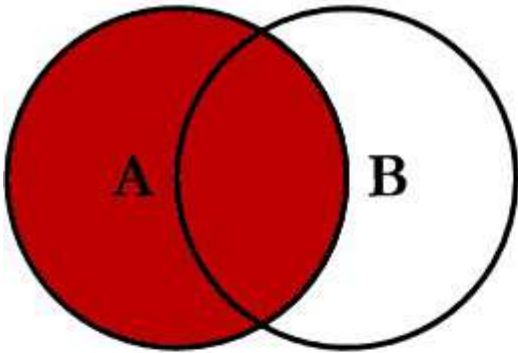
```
SELECT videogame.id,
       videogame.title,
       videogame.reg_date,
       tags.name,
       tags_meta.post_id
FROM tags_meta
INNER JOIN videogame ON videogame.id = tags_meta.post_id
INNER JOIN tags ON tags.id = tags_meta.tag_id
WHERE tags.name = "elizabeth"
ORDER BY videogame.reg_date
```

this code can return all posts which related to that tag "#elizabeth"

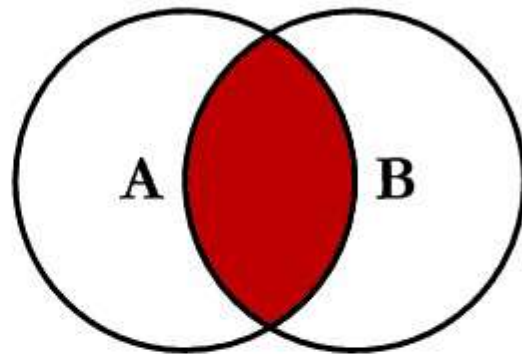
Joins visualized

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

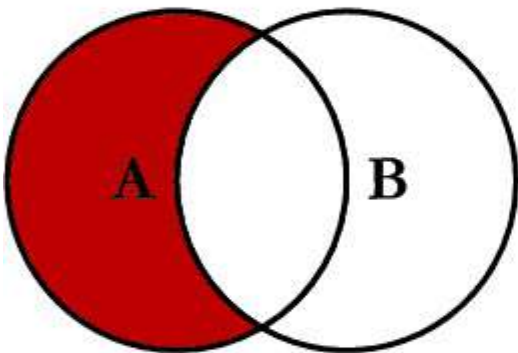
SQL JOINS



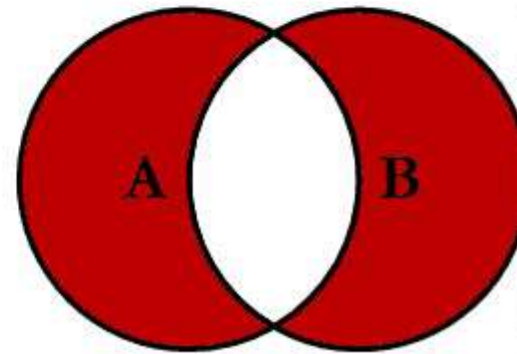
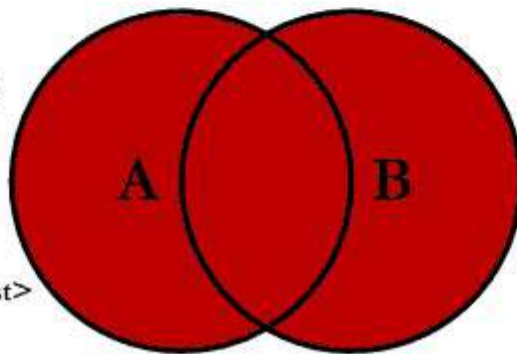
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```

© C.L. Moffatt, 2008

Read Joins online: <https://riptutorial.com/mysql/topic/2736/joins>

Chapter 34: JOINS: Join 3 table with the same name of id.

Examples

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
```

Read JOINS: Join 3 table with the same name of id. online:

<https://riptutorial.com/mysql/topic/9921/joins--join-3-table-with-the-same-name-of-id->

Chapter 35: JSON

Introduction

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>

Remarks

Starting from MySQL 5.7.8, MySQL ships with a JSON type. Lots of devs have been saving JSON data in text columns for a long time but the JSON type is different, the data is saved in binary format after validation. That avoids the overhead of parsing the text on each read.

Examples

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)  
);
```

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.

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.

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 explanation 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)
```

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);
```

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 ;
```

Read JSON online: <https://riptutorial.com/mysql/topic/2985/json>

Chapter 36: Limit and Offset

Syntax

- `SELECT column_1 [, column_2]`
`FROM table_1`
`ORDER BY order_column`
`LIMIT row_count [OFFSET row_offset]`
- `SELECT column_1 [, column_2]`
`FROM table_1`
`ORDER BY order_column`
`LIMIT [row_offset,] row_count`

Remarks

"Limit" could mean "Max number of rows in a table".

"Offset" mean pick from `row` number (not to be confused by primary key value or any field data value)

Examples

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.

Read Limit and Offset online: <https://riptutorial.com/mysql/topic/548/limit-and-offset>

Chapter 37: LOAD DATA INFILE

Syntax

1. LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'
2. INTO TABLE tbl_name
3. [CHARACTER SET charset]
4. [{FIELDS | COLUMNS} [TERMINATED BY 'string'] [[OPTIONALLY] ENCLOSED BY 'char']]
5. [LINES [STARTING BY 'string'] [TERMINATED BY 'string']]
6. [IGNORE number {LINES | ROWS}]
7. [(col_name_or_user_var,...)]
8. [SET col_name = expr,...]

Examples

using LOAD DATA INFILE to load large amount of data to database

Consider the following example assuming that you have a ';' delimited CSV to load into your database.

```
1;max;male;manager;12-7-1985
2;jack;male;executive;21-8-1990
.
.
.
1000000;marta;female;accountant;15-6-1992
```

Create the table for insertion.

```
CREATE TABLE `employee` ( `id` INT NOT NULL ,
                           `name` VARCHAR NOT NULL,
                           `sex` VARCHAR NOT NULL ,
                           `designation` VARCHAR NOT NULL ,
                           `dob` VARCHAR NOT NULL );
```

Use the following query to insert the values in that table.

```
LOAD DATA INFILE 'path of the file/file_name.txt'
INTO TABLE employee
FIELDS TERMINATED BY ';' //specify the delimiter separating the values
LINES TERMINATED BY '\r\n'
(id,name,sex,designation,dob)
```

Consider the case where the date format is non standard.

```
1;max;male;manager;17-Jan-1985
2;jack;male;executive;01-Feb-1992
.
```

```
.  
.1000000;marta;female;accountant;25-Apr-1993
```

In this case you can change the format of the `dob` column before inserting like this.

```
LOAD DATA INFILE 'path of the file/file_name.txt'  
INTO TABLE employee  
FIELDS TERMINATED BY ';' //specify the delimiter separating the values  
LINES TERMINATED BY '\r\n'  
(id,name,sex,designation,@dob)  
SET date = STR_TO_DATE(@date, '%d-%b-%Y');
```

This example of `LOAD DATA INFILE` does not specify all the available features.

You can see more references on `LOAD DATA INFILE` [here](#).

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
```

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 ...
```

import / export

import

```
SELECT a,b,c INTO OUTFILE 'result.txt' FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'  
LINES TERMINATED BY '\n' FROM table;
```

Export

```
LOAD DATA INFILE 'result.txt' INTO TABLE table;
```

Read `LOAD DATA INFILE` online: <https://riptutorial.com/mysql/topic/2356/load-data-infile>

Chapter 38: Log files

Examples

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.)

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

```

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:

```

36 Query insert questions_c23(qId,ownerId,title,votes,answers,isClosed,closeVotes,views,owne
comments,answeredAccepted,askDate,closeDate,lastScanDate,ign,bn,pvtc,
mainTagForImport,prepStatus,touches,status,status_bef_change,cv_bef_change,max_cv_r
values(38666373, 1322183, 'How to post a numeric value in c#', 0, 1, 0, 0, 50, 1,
0, 0, '2016-07-29 19:40:32', null, now(), 0, 0, 0,
'c%23',0,1,'0',' ',0,0)
on duplicate key update title='How to post a numeric value in c#', votes=0, answers
answeredAccepted=0,lastScanDate=now(), touches=touches+1,status='0'

```

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](#).

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:

```
2016-08-02 20:40:39 2420 [Note] Shutting down plugin 'binlog'
2016-08-02 20:40:39 2420 [Note] mysqld: Shutdown complete

2016-08-02 20:43:11 2888 [Note] Plugin 'FEDERATED' is disabled.
2016-08-02 20:43:11 2888 [Note] InnoDB: Using atomics to ref count buffer pool pages
2016-08-02 20:43:11 2888 [Note] InnoDB: The InnoDB memory heap is disabled
```

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`.

Read Log files online: <https://riptutorial.com/mysql/topic/5102/log-files>

Chapter 39: Many-to-many Mapping table

Remarks

- Lack of an `AUTO_INCREMENT` id for this table -- The PK given is the 'natural' PK; there is no good reason for a surrogate.
- `MEDIUMINT` -- This is a reminder that all `INT`s should be made as small as is safe (smaller \Rightarrow faster). Of course the declaration here must match the definition in the table being linked to.
- `UNSIGNED` -- Nearly all `INT`s may as well be declared non-negative
- `NOT NULL` -- Well, that's true, isn't it?
- `InnoDB` -- More efficient than `MyISAM` because of the way the `PRIMARY KEY` is clustered with the data in `InnoDB`.
- `INDEX(y_id, x_id)` -- The `PRIMARY KEY` makes it efficient to go one direction; the makes the other direction efficient. No need to say `UNIQUE`; that would be extra effort on `INSERT`s.
- In the secondary index, saying just `INDEX(y_id)` would work because it would implicit include `x_id`. But I would rather make it more obvious that I am hoping for a 'covering' index.

You *may* want to add more columns to the table; this is rare. The extra columns could provide information about the *relationship* that the table represents.

You *may* want to add `FOREIGN KEY` constraints.

Examples

Typical schema

```
CREATE TABLE XtoY (  
  # No surrogate id for this table  
  x_id MEDIUMINT UNSIGNED NOT NULL,    -- For JOINing to one table  
  y_id MEDIUMINT UNSIGNED NOT NULL,    -- For JOINing to the other table  
  # Include other fields specific to the 'relation'  
  PRIMARY KEY(x_id, y_id),              -- When starting with X  
  INDEX      (y_id, x_id)                -- When starting with Y  
) ENGINE=InnoDB;
```

(See Remarks, below, for rationale.)

Read Many-to-many Mapping table online: <https://riptutorial.com/mysql/topic/4857/many-to-many-mapping-table>

Chapter 40: MyISAM Engine

Remarks

Over the years, InnoDB has improved to the point where it is almost always better than MyISAM, at least the currently supported versions. Bottom line: Don't use MyISAM, except maybe for tables that are truly temporary.

One advantage of MyISAM over InnoDB remains: It is 2x-3x smaller in space required on disk.

When InnoDB first came out, MyISAM was still a viable Engine. But with the advent of XtraDB and 5.6, InnoDB became "better" than MyISAM in most benchmarks.

Rumor has it that the next major version will eliminate the need for MyISAM by making truly temporary InnoDB tables and by moving the system tables into InnoDB.

Examples

ENGINE=MyISAM

```
CREATE TABLE foo (  
    ...  
) ENGINE=MyISAM;
```

Read MyISAM Engine online: <https://riptutorial.com/mysql/topic/4710/myisam-engine>

Chapter 41: MySQL Admin

Examples

Change root password

```
mysqladmin -u root -p'old-password' password 'new-password'
```

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
```

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;
```

Read MySQL Admin online: <https://riptutorial.com/mysql/topic/2991/mysql-admin>

Chapter 42: MySQL client

Syntax

- mysql [OPTIONS] [database_name]

Parameters

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 <i>Note: there is no space between -p and the password</i>
-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 <code>sql_safe_updates=ON</code> . This will allow only <code>DELETE</code> and <code>UPDATE</code> that explicitly use keys
-V --version	print the version and exit

Examples

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.

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
```

Read MySQL client online: <https://riptutorial.com/mysql/topic/5619/mysql-client>

Chapter 43: MySQL LOCK TABLE

Syntax

- `LOCK TABLES table_name [READ | WRITE];` // Lock Table
- `UNLOCK TABLES;` // Unlock Tables

Remarks

Locking is used to solve concurrency problems. Locking is required only when running a transaction, that first read a value from a database and later write that value in to the database. Locks are never required for self-contained insert, update, or delete operations.

There are two kinds of locks available

READ LOCK - when a user is only reading from a table.

WRITE LOCK - when a user is doing both reading and writing to a table.

When a user holds a `WRITE LOCK` on a table, no other users can read or write to that table. When a user holds a `READ LOCK` on a table, other users can also read or hold a `READ LOCK`, but no user can write or hold a `WRITE LOCK` on that table.

If default storage engine is 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.

For all storage engines other than InnoDB, MySQL uses table locking.

For more details about table lock [See here](#)

Examples

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

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 0xf996beac700, 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 committed 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.

Read MySQL LOCK TABLE online: <https://riptutorial.com/mysql/topic/5233/mysql-lock-table>

Chapter 44: Mysql Performance Tips

Examples

Select Statement Optimization

Below are some tips to remember while we are writing a select query in MySQL that can help us and reduce our query time:-

1. Whenever we use where in a large table we should make sure the column in where clause are index or not. Ex:- Select * from employee where user_id > 2000. user_id if indexed then will speed up the evaluation of the query alot. Indexes are also very important during joins and foreign keys.
2. When you need the smaller section of content rather then fetching whole data from table, try to use limit. Rather then writing Ex:- Select * from employee. If you need just first 20 employee from lakhs then just use limit Ex:- Select * from employee LIMIT 20.
3. You can also optimize your query by providing the column name which you want in resultset. Rather then writing Ex:- Select * from employee. Just mention column name from which you need data if you table has lots of column and you want to have data for few of them. Ex:- Select id, name from employee.
4. Index column if you are using to verify for NULL in where clause. If you have some statement as SELECT * FROM tbl_name WHERE key_col IS NULL; then if key_col is indexed then query will be evaluated faster.

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.

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 .

Read MySQL Performance Tips online: <https://riptutorial.com/mysql/topic/5752/mysql-performance-tips>

Chapter 45: MySQL Unions

Syntax

- `SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;`
- `SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;`
- `SELECT column_name(s) FROM table1 WHERE col_name="XYZ" UNION ALL SELECT column_name(s) FROM table2 WHERE col_name="XYZ";`

Remarks

`UNION DISTINCT` is the same as `UNION`; it is slower than `UNION ALL` because of a de-duplicating pass. A good practice is to always spell out `DISTINCT` or `ALL`, thereby signaling that you thought about which to do.

Examples

Union operator

The `UNION` operator is used to combine the result-set (*only distinct values*) of two or more `SELECT` statements.

Query: (To select 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
```


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
```

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

Read MySQL Unions online: <https://riptutorial.com/mysql/topic/5376/mysql-unions>

Chapter 46: mysqlimport

Parameters

Parameter	Description
<code>--delete -D</code>	empty the table before importing the text file
<code>--fields-optionally-enclosed-by</code>	define the character that quotes the fields
<code>--fields-terminated-by</code>	field terminator
<code>--ignore -i</code>	ignore the ingested row in case of duplicate-keys
<code>--lines-terminated-by</code>	define row terminator
<code>--password -p</code>	password
<code>--port -P</code>	port
<code>--replace -r</code>	overwrite the old entry row in case of duplicate-keys
<code>--user -u</code>	username
<code>--where -w</code>	specify a condition

Remarks

`mysqlimport` will use the name of the imported file, after stripping the extension, to determine the destination table.

Examples

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
```

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
```

Using a custom row-delimiter

This example is useful for windows-like endings:

```
$ mysqlimport --lines-terminated-by='\r\n' mycompany employee.txt
```

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

Conditional import

```
$ mysqlimport --where="id>2" mycompany employee.txt
```

Import a standard csv

```
$ mysqlimport
  --fields-optionally-enclosed-by='''
  --fields-terminated-by=,
  --lines-terminated-by="\r\n"
mycompany employee.csv
```

Read `mysqlimport` online: <https://riptutorial.com/mysql/topic/5215/mysqlimport>

Chapter 47: NULL

Examples

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?

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
```

Read NULL online: <https://riptutorial.com/mysql/topic/6757/null>

Chapter 48: One to Many

Introduction

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.

Remarks

For most cases, working with a 1:M relationship requires us to understand *Primary Keys* and *Foreign Keys*.

A Primary key is a column in a table where any single row of that column represents a single entity, or, selecting a value in a primary key column results in exactly one row. Using the above examples, an EMP_ID represents a single employee. If you query for any single EMP_ID, you will see a single row representing the corresponding employee.

A Foreign Key is a column in a table that corresponds to the primary key of another different table. From our example above, the MGR_ID in the EMPLOYEES table is a foreign key. Generally to join two tables, you'll join them based on the primary key of one table and the foreign key in another.

Examples

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

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.

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.

Read One to Many online: <https://riptutorial.com/mysql/topic/9600/one-to-many>

Chapter 49: ORDER BY

Examples

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
```

Basic

ORDER BY x

x can be any datatype.

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

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`
- `NULLs` 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.

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.

Read ORDER BY online: <https://riptutorial.com/mysql/topic/5469/order-by>

Chapter 50: Partitioning

Remarks

- **RANGE partitioning.** This type of partitioning assigns rows to partitions based on column values falling within a given range.
- **LIST partitioning.** Similar to partitioning by RANGE, except that the partition is selected based on columns matching one of a set of discrete values.
- **HASH partitioning.** With this type of partitioning, a partition is selected based on the value returned by a user-defined expression that operates on column values in rows to be inserted into the table. The function may consist of any expression valid in MySQL that yields a nonnegative integer value. An extension to this type, `LINEAR HASH`, is also available.
- **KEY partitioning.** This type of partitioning is similar to partitioning by HASH, except that only one or more columns to be evaluated are supplied, and the MySQL server provides its own hashing function. These columns can contain other than integer values, since the hashing function supplied by MySQL guarantees an integer result regardless of the column data type. An extension to this type, `LINEAR KEY`, is also available.

Examples

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](#).

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](#).

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](#).

Read Partitioning online: <https://riptutorial.com/mysql/topic/5128/partitioning>

Chapter 51: Performance Tuning

Syntax

- Don't use DISTINCT and GROUP BY in the same SELECT.
- Don't paginate via OFFSET, "remember where you left off".
- WHERE (a,b) = (22,33) does not optimize at all.
- Explicitly say ALL or DISTINCT after UNION -- it reminds you pick between the faster ALL or the slower DISTINCT.
- Don't use SELECT *, especially if you have TEXT or BLOB columns that you don't need. There is overhead in tmp tables and transmission.
- It is faster when the GROUP BY and ORDER BY can have exactly the same list.
- Don't use FORCE INDEX; it may help today, but will probably hurt tomorrow.

Remarks

See also discussions about ORDER BY, LIKE, REGEXP, etc. Note: this needs editing with links and more Topics.

[Cookbook on building optimal indexes.](#)

Examples

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
```

Set the cache correctly

`innodb_buffer_pool_size` should be about 70% of available RAM.

Avoid inefficient constructs

```
x IN ( SELECT ... )
```

turn into a `JOIN`

When possible, avoid `OR`.

Do not 'hide' an indexed column in a function, such as `WHERE DATE(x) = ...`; reformulate as `WHERE x = ...`

You can generally avoid `WHERE LCASE(name1) = LCASE(name2)` by having a suitable collation.

Do not use `OFFSET` for "pagination", instead 'remember where you left off'.

Avoid `SELECT *...` (unless debugging).

Note to Maria Deleva, Barranka, Batsu: This is a place holder; please make remove these items as you build full-scale examples. After you have done the ones you can, I will move in to elaborate on the rest and/or toss them.

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.)

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
```

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
```

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.

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 of 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`.

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`.

Read Performance Tuning online: <https://riptutorial.com/mysql/topic/4292/performance-tuning>

Chapter 52: Pivot queries

Remarks

Pivot query creation in MySQL relies upon the `GROUP_CONCAT()` function. If the result of the expression that creates the columns of the pivot query is expected to be large, the value of the `group_concat_max_len` variable must be increased:

```
set session group_concat_max_len = 1024 * 1024; -- This should be enough for most cases
```

Examples

Creating a pivot query

MySQL does not provide a built-in way to create pivot queries. However, these can be created using prepared statements.

Assume the table `tbl_values`:

Id	Name	Group	Value
1	Pete	A	10
2	Pete	B	20
3	John	A	10

Request: Create a query that shows the sum of `Value` for each `Name`; the `Group` must be column header and `Name` must be the row header.

```
-- 1. Create an expression that builds the columns
set @sql = (
    select group_concat(distinct
        concat(
            "sum(case when `Group`='", Group, "' then `Value` end) as `", `Group`, "`"
        )
    )
    from tbl_values
);

-- 2. Complete the SQL instruction
set @sql = concat("select Name, ", @sql, " from tbl_values group by `Name`");

-- 3. Create a prepared statement
prepare stmt from @sql;

-- 4. Execute the prepared statement
execute stmt;
```

Result:

Name	A	B
John	10	NULL
Pete	10	20

Important: Deallocate the prepared statement once it's no longer needed:

```
deallocate prepare stmt;
```

[Example on SQL Fiddle](#)

Read Pivot queries online: <https://riptutorial.com/mysql/topic/3074/pivot-queries>

Chapter 53: PREPARE Statements

Syntax

- `PREPARE stmt_name FROM preparable_stmt`
- `EXECUTE stmt_name [USING @var_name [, @var_name] ...]`
- `{DEALLOCATE | DROP} PREPARE stmt_name`

Examples

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.

Construct and execute

(This is a request for a good example that shows how to *construct* a `SELECT` using `CONCAT`, then prepare+execute it. Please emphasize the use of `@variables` versus `DECLARED` variables -- it makes a big difference, and it is something that novices (include myself) stumble over.)

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;
```

Read PREPARE Statements online: <https://riptutorial.com/mysql/topic/2603/prepare-statements>

Chapter 54: Recover and reset the default root password for MySQL 5.7+

Introduction

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.

Remarks

Recovering and resetting the default root password using this method is applicable only for MySQL 5.7+

Examples

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.

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.

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>
```

Read Recover and reset the default root password for MySQL 5.7+ online:

<https://riptutorial.com/mysql/topic/9563/recover-and-reset-the-default-root-password-for-mysql-5-7plus>

Chapter 55: Recover from lost root password

Examples

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
```

Read Recover from lost root password online: <https://riptutorial.com/mysql/topic/9973/recover-from-lost-root-password>

Chapter 56: Regular Expressions

Introduction

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

Examples

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
```

Read Regular Expressions online: <https://riptutorial.com/mysql/topic/9444/regular-expressions>

Chapter 57: Replication

Remarks

Replication is used to copy[Backup] data from one MySQL database server to one or more MySQL database servers.

Master -- The MySQL database server, which is serving data to be copied

Slave -- The MySQL database server, copies data which is served by Master

With MySQL, replication is asynchronous by default. This means slaves do not need to be connected permanently to receive updates from the master. For example, if your slave is switched OFF or not connected with master and you are switching slave ON or connect with Master at a later time, then it will automatically synchronize with the Master.

Depending on the configuration, you can replicate all databases, selected databases, or even selected tables within a database.

Replication Formats

There are two core types of replication formats

Statement Based Replication (SBR) -- which replicates entire SQL statements. In this, the master writes SQL statements to the binary log. Replication of the master to the slave works by executing that SQL statements on the slave.

Row Based Replication (RBR) -- which replicates only the changed rows. In this, the master writes events to the binary log that indicate how individual table rows are changed. Replication of the master to the slave works by copying the events representing the changes to the table rows to the slave.

You can also use a third variety, *Mixed Based Replication (MBR)*. In this, both statement-based and row-based logging is used. Log will be created depending on which is most appropriate for the change.

Statement-based format was the default in MySQL versions older than 5.7.7. In MySQL 5.7.7 and later, row-based format is the default.

Examples

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 will 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.

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
```

Read Replication online: <https://riptutorial.com/mysql/topic/7218/replication>

Chapter 58: Reserved Words

Introduction

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.

Remarks

Listed below are all reserved words (from [the official documentation](#)):

- ACCESSIBLE
- ADD
- ALL
- ALTER
- ANALYZE
- AND
- AS
- ASC
- ASENSITIVE
- BEFORE
- BETWEEN
- BIGINT
- BINARY
- BLOB
- BOTH
- BY
- CALL
- CASCADE
- CASE
- CHANGE
- CHAR
- CHARACTER
- CHECK
- COLLATE
- COLUMN
- CONDITION
- CONSTRAINT
- CONTINUE
- CONVERT
- CREATE

- CROSS
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- CURSOR
- DATABASE
- DATABASES
- DAY_HOUR
- DAY_MICROSECOND
- DAY_MINUTE
- DAY_SECOND
- DEC
- DECIMAL
- DECLARE
- DEFAULT
- DELAYED
- DELETE
- DESC
- DESCRIBE
- DETERMINISTIC
- DISTINCT
- DISTINCTROW
- DIV
- DOUBLE
- DROP
- DUAL
- EACH
- ELSE
- ELSEIF
- ENCLOSED
- ESCAPED
- EXISTS
- EXIT
- EXPLAIN
- FALSE
- FETCH
- FLOAT
- FLOAT4
- FLOAT8
- FOR
- FORCE
- FOREIGN
- FROM
- FULLTEXT
- GENERATED

- GET
- GRANT
- GROUP
- HAVING
- HIGH_PRIORITY
- HOUR_MICROSECOND
- HOUR_MINUTE
- HOUR_SECOND
- IF
- IGNORE
- IN
- INDEX
- INFILE
- INNER
- INOUT
- INSENSITIVE
- INSERT
- INT
- INT1
- INT2
- INT3
- INT4
- INT8
- INTEGER
- INTERVAL
- INTO
- IO_AFTER_GTIDS
- IO_BEFORE_GTIDS
- IS
- ITERATE
- JOIN
- KEY
- KEYS
- KILL
- LEADING
- LEAVE
- LEFT
- LIKE
- LIMIT
- LINEAR
- LINES
- LOAD
- LOCALTIME
- LOCALTIMESTAMP
- LOCK
- LONG

- LONGBLOB
- LONGTEXT
- LOOP
- LOW_PRIORITY
- MASTER_BIND
- MASTER_SSL_VERIFY_SERVER_CERT
- MATCH
- MAXVALUE
- MEDIUMBLOB
- MEDIUMINT
- MEDIUMTEXT
- MIDDLEINT
- MINUTE_MICROSECOND
- MINUTE_SECOND
- MOD
- MODIFIES
- NATURAL
- NOT
- NO_WRITE_TO_BINLOG
- NULL
- NUMERIC
- ON
- OPTIMIZE
- OPTIMIZER_COSTS
- OPTION
- OPTIONALLY
- OR
- ORDER
- OUT
- OUTER
- OUTFILE
- PARTITION
- PRECISION
- PRIMARY
- PROCEDURE
- PURGE
- RANGE
- READ
- READS
- READ_WRITE
- REAL
- REFERENCES
- REGEXP
- RELEASE
- RENAME
- REPEAT

- REPLACE
- REQUIRE
- RESIGNAL
- RESTRICT
- RETURN
- REVOKE
- RIGHT
- RLIKE
- SCHEMA
- SCHEMAS
- SECOND_MICROSECOND
- SELECT
- SENSITIVE
- SEPARATOR
- SET
- SHOW
- SIGNAL
- SMALLINT
- SPATIAL
- SPECIFIC
- SQL
- SQLEXCEPTION
- SQLSTATE
- SQLWARNING
- SQL_BIG_RESULT
- SQL_CALC_FOUND_ROWS
- SQL_SMALL_RESULT
- SSL
- STARTING
- STORED
- STRAIGHT_JOIN
- TABLE
- TERMINATED
- THEN
- TINYBLOB
- TINYINT
- TINYTEXT
- TO
- TRAILING
- TRIGGER
- TRUE
- UNDO
- UNION
- UNIQUE
- UNLOCK
- UNSIGNED

- UPDATE
- USAGE
- USE
- USING
- UTC_DATE
- UTC_TIME
- UTC_TIMESTAMP
- VALUES
- VARBINARY
- VARCHAR
- VARCHARACTER
- VARYING
- VIRTUAL
- WHEN
- WHERE
- WHILE
- WITH
- WRITE
- XOR
- YEAR_MONTH
- ZEROFILL
- GENERATED
- OPTIMIZER_COSTS
- STORED
- VIRTUAL

Examples

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.](#)

Read Reserved Words online: <https://riptutorial.com/mysql/topic/1398/reserved-words>

Chapter 59: Security via GRANTS

Examples

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 dbname.* ...      -- "read only"
GRANT ... ON dbname.tblname ...    -- "just one table"
```

The readonly may also need 'safe' things like

```
GRANT SELECT, CREATE TEMPORARY TABLE ON dbname.* ...      -- "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.

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`.

Read Security via GRANTs online: <https://riptutorial.com/mysql/topic/5131/security-via-grants>

Chapter 60: SELECT

Introduction

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

Syntax

- `SELECT DISTINCT [expressions] FROM TableName [WHERE conditions];` ///Simple Select
- `SELECT DISTINCT(a), b ...` is the same as `SELECT DISTINCT a, b ...`
- `SELECT [ALL | DISTINCT | DISTINCTROW] [HIGH_PRIORITY] [STRAIGHT_JOIN] [SQL_SMALL_RESULT | SQL_BIG_RESULT] [SQL_BUFFER_RESULT] [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS] expressions FROM tables [WHERE conditions] [GROUP BY expressions] [HAVING condition] [ORDER BY expression [ASC | DESC]] [LIMIT [offset_value] number_rows | LIMIT number_rows OFFSET offset_value] [PROCEDURE procedure_name] [INTO [OUTFILE 'file_name' options | DUMPFILE 'file_name' | @variable1, @variable2, ... @variable_n] [FOR UPDATE | LOCK IN SHARE MODE];` ///Full Select Syntax

Remarks

For more information on MySQL's `SELECT` statement, refer [MySQL Docs](#).

Examples

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 |
+-----+
```

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

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.

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`.

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)
```

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.

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](#).

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   |
+-----+
```

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'

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.

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`).

Read `SELECT` online: <https://riptutorial.com/mysql/topic/3307/select>

Chapter 61: Server Information

Parameters

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.

Examples

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;
```

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.

Read Server Information online: <https://riptutorial.com/mysql/topic/9924/server-information>

Chapter 62: SSL Connection Setup

Examples

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>

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 firewall

```
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.

Read SSL Connection Setup online: <https://riptutorial.com/mysql/topic/7563/ssl-connection-setup>

Chapter 63: Stored routines (procedures and functions)

Parameters

Parameter	Details
RETURNS	Specifies the data type that can be returned from a function.
RETURN	Actual variable or value following the <code>RETURN</code> syntax is what is returned to where the function was called from.

Remarks

A stored routine is either a procedure or a function.

A procedure is invoked using a `CALL` statement and can only pass back values using output variables.

A function can be called from inside a statement just like any other function and can return a scalar value.

Examples

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.

Create Procedure with a Constructed Prepare

```
DROP PROCEDURE if exists displayNext100WithName;
DELIMITER $$
CREATE PROCEDURE displayNext100WithName
(
  nStart int,
  tblName varchar(100)
)
BEGIN
  DECLARE thesql varchar(500); -- holds the constructed sql string to execute

  -- expands the sizing of the output buffer to accomodate the output (Max value is at least
  4GB)
  SET session group_concat_max_len = 4096; -- prevents group_concat from barfing with error
  1160 or whatever it is

  SET @thesql=CONCAT("select group_concat(qid order by qid SEPARATOR '%3B') as nums ","from
  (
    select qid from ");
  SET @thesql=CONCAT(@thesql,tblName," where qid>? order by qid limit 100 )xDerived");
  PREPARE stmt1 FROM @thesql; -- create a statement object from the construct sql string to
```

```

execute
    SET @p1 = nStart; -- transfers parameter passed into a User Variable compatible with the
below EXECUTE
    EXECUTE stmt1 USING @p1;

    DEALLOCATE PREPARE stmt1; -- deallocate the statement object when finished
END$$
DELIMITER ;

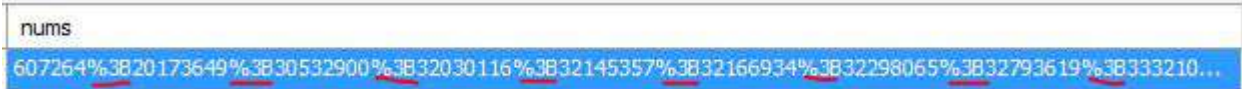
```

Creation of the stored procedure shows wrapping with a DELIMITER necessary in many client tools.

Calling example:

```
call displayNext100WithName(1, "questions_mysql");
```

Sample output with %3B (semi-colon) separator:



nums
607264%3B20173649%3B30532900%3B32030116%3B32145357%3B32166934%3B32298065%3B32793619%3B333210...

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>

Cursors

Cursors enable you to iterate 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 student;
```

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.

Multiple ResultSets

Unlike a `SELECT` statement, a `Stored Procedure` returns multiple result sets. This requires different code to be used for gathering the results of a `CALL` in Perl, PHP, etc.

(Need specific code here or elsewhere!)

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.

Read [Stored routines \(procedures and functions\)](https://riptutorial.com/mysql/topic/1351/stored-routines--procedures-and-functions-) online:

<https://riptutorial.com/mysql/topic/1351/stored-routines--procedures-and-functions->

Chapter 64: String operations

Parameters

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

Name	Description
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

Name	Description
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

Examples

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

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.

LOWER() / LCASE()

Convert in lowercase the string argument

Syntax: LOWER(str)

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

REPLACE()

Convert in lowercase the string argument

Syntax: REPLACE(str, from_str, to_str)

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

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'
```

UPPER() / UCASE()

Convert in uppercase the string argument

Syntax: UPPER(str)

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

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
```

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
```

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
```

Read String operations online: <https://riptutorial.com/mysql/topic/1399/string-operations>

Chapter 65: Table Creation

Syntax

- `CREATE TABLE table_name (column_name1 data_type(size), column_name2 data_type(size), column_name3 data_type(size),);` // Basic table creation
- `CREATE TABLE table_name [IF NOT EXISTS] (column_name1 data_type(size), column_name2 data_type(size), column_name3 data_type(size),);` // Table creation checking existing
- `CREATE [TEMPORARY] TABLE table_name [IF NOT EXISTS] (column_name1 data_type(size), column_name2 data_type(size), column_name3 data_type(size),);` // Temporary table creation
- `CREATE TABLE new_tbl [AS] SELECT * FROM orig_tbl;` // Table creation from SELECT

Remarks

The `CREATE TABLE` statement should end with an `ENGINE` specification:

```
CREATE TABLE table_name ( column_definitions ) ENGINE=engine;
```

Some options are:

- **InnoDB:** (Default since version 5.5.5) It's a transaction-safe (ACID compliant) engine. It has transaction commit and roll-back, and crash-recovery capabilities and row-level locking.
- **MyISAM:** (Default before version 5.5.5) It's a plain-fast engine. It doesn't support transactions, nor foreign keys, but it's useful for data-warehousing.
- **Memory:** Stores all data in RAM for extremely fast operations but table data will be lost on database restart.

More engine options [here](#).

Examples

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 enclose 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.

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`.

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.)

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;
```

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:

```
-- 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:

```
-- 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:

```
FROM NAME_DATABASE.name_table
```

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](#).

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'
;
```

Read Table Creation online: <https://riptutorial.com/mysql/topic/795/table-creation>

Chapter 66: Temporary Tables

Examples

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;
```

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

Read Temporary Tables online: <https://riptutorial.com/mysql/topic/5757/temporary-tables>

Chapter 67: Time with subsecond precision

Remarks

You need to be at MySQL version 5.6.4 or later to declare columns with fractional-second time datatypes.

For example, `DATETIME(3)` will give you millisecond resolution in your timestamps, and `TIMESTAMP(6)` will give you microsecond resolution on a *nix-style timestamp.

Read this: <http://dev.mysql.com/doc/refman/5.7/en/fractional-seconds.html>

`NOW(3)` will give you the present time from your MySQL server's operating system with millisecond precision.

(Notice that MySQL internal fractional arithmetic, like `* 0.001`, is always handled as IEEE754 double precision floating point, so it's unlikely you'll lose precision before the Sun becomes a white dwarf star.)

Examples

Get the current time with millisecond precision

```
SELECT NOW(3)
```

does the trick.

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.

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.

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.

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.)

Read Time with subsecond precision online: <https://riptutorial.com/mysql/topic/7850/time-with-subsecond-precision>

Chapter 68: Transaction

Examples

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.

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 autocommit false
SET AUTOCOMMIT=false;
--or
SET AUTOCOMMIT=0;

-->To make autocommit 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` is 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

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&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&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.

Read Transaction online: <https://riptutorial.com/mysql/topic/5771/transaction>

Chapter 69: TRIGGERS

Syntax

- `CREATE [DEFINER = { user | CURRENT_USER }] TRIGGER trigger_name trigger_time trigger_event ON tbl_name FOR EACH ROW [trigger_order] trigger_body`
- `trigger_time`: { BEFORE | AFTER }
- `trigger_event`: { INSERT | UPDATE | DELETE }
- `trigger_order`: { FOLLOWS | PRECEDES } other_trigger_name

Remarks

Two points need to draw your attention if you already use triggers on others DB :

FOR EACH ROW

FOR EACH ROW is a mandatory part of the syntax

You can't make a *statement* trigger (once by query) like Oracle do. It's more a performance related issue than a real missing feature

CREATE OR REPLACE TRIGGER

The `CREATE OR REPLACE` is not supported by MySQL

MySQL don't allow this syntax, you have instead to use the following :

```
DELIMITER $$

DROP TRIGGER IF EXISTS myTrigger;
$$
CREATE TRIGGER myTrigger
-- ...

$$
DELIMITER ;
```

Be careful, this is **not an atomic transaction** :

- you'll loose the old trigger if the `CREATE` fail
- on a heavy load, others operations can occurs between the `DROP` and the `CREATE`, use a `LOCK TABLES myTable WRITE; first to avoid data inconsistency and UNLOCK TABLES; after the CREATE to release the table`

Examples

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.

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 ;
```

Read TRIGGERS online: <https://riptutorial.com/mysql/topic/3069/triggers>

Chapter 70: UNION

Syntax

- UNION DISTINCT -- dedups after combining the SELECTs
- UNION ALL -- non dedup (faster)
- UNION -- the default is DISTINCT
- SELECT ... UNION SELECT ... -- is OK, but ambiguous on ORDER BY
- (SELECT ...) UNION (SELECT ...) ORDER BY ... -- resolves the ambiguity

Remarks

UNION does not use multiple CPUs.

UNION always* involves a temp table to collect the results. *As of 5.7.3 / MariaDB 10.1, some forms of UNION deliver the results without using a tmp table (hence, faster).

Examples

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`.

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.

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`.

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.

UNION ALL and UNION

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

信息	结果1	概况	状态
1	22	44	
▶ 1	22	44	
	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

信息	结果1	概况	状态
1	22	44	
▶ 1	22	44	
2	33	55	
2	33	55	

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);
```

Read UNION online: <https://riptutorial.com/mysql/topic/3847/union>

Chapter 71: UPDATE

Syntax

- UPDATE [LOW_PRIORITY] [IGNORE] tableName SET column1 = expression1, column2 = expression2, ... [WHERE conditions]; //Simple single table update
- UPDATE [LOW_PRIORITY] [IGNORE] tableName SET column1 = expression1, column2 = expression2, ... [WHERE conditions] [ORDER BY expression [ASC | DESC]] [LIMIT row_count]; //Update with order by and limit
- UPDATE [LOW_PRIORITY] [IGNORE] table1, table2, ... SET column1 = expression1, column2 = expression2, ... [WHERE conditions]; //Multiple Table update

Examples

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

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.

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`.

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.

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.

Read UPDATE online: <https://riptutorial.com/mysql/topic/2738/update>

Chapter 72: Using Variables

Examples

Setting Variables

Here are some ways to set variables:

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

EX: 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 :=

EX: 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)

EX: 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;
```

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 |
+-----+-----+-----+
```

Read Using Variables online: <https://riptutorial.com/mysql/topic/5013/using-variables>

Chapter 73: VIEW

Syntax

- `CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition; ///` Simple create view syntax
- `CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] [DEFINER = { user | CURRENT_USER }] [SQL SECURITY { DEFINER | INVOKER }] VIEW view_name [(column_list)] AS select_statement [WITH [CASCADED | LOCAL] CHECK OPTION]; ///` Full Create view syntax
- `DROP VIEW [IF EXISTS] [db_name.]view_name; ///` Drop view syntax

Parameters

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.

Remarks

Views are virtual tables and do not contain the data that is returned. They can save you from writing complex queries again and again.

- **Before a view is made** its specification consists entirely of a `SELECT` statement. The `SELECT` statement cannot contain a sub-query in the `FROM` clause.
- **Once a view is made** it is used largely just like a table and can be `SELECTED` from just like a table.

You have to create views, when you want to restrict few columns of your table, from the other user.

- For example: In your organization, you want your managers to view few information from a table named-"Sales", but you don't want that your software engineers can view all fields of table-"Sales". Here, you can create two different views for your managers and your software engineers.

Performance. `VIEWS` are syntactic sugar. However there performance may or may not be worse than the equivalent query with the view's select folded in. The Optimizer attempts to do this "fold in" for you, but is not always successful. MySQL 5.7.6 provides some more enhancements in the Optimizer. But, regardless, using a `VIEW` will not generate a *faster* query.

Examples

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.

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

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](#).

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;
```

Read VIEW online: <https://riptutorial.com/mysql/topic/1489/view>

Credits

S. No	Chapters	Contributors
1	Getting started with MySQL	A. Raza , Aman Dhanda , Andy , Athafoud , CodeWarrior , Community , Configure , Dipen Shah , e4c5 , Epodax , Giacomo Garabello , greatwolf , inetphantom , JayRizzo , juergen d , Lahiru Ashan , Lambda Ninja , Magisch , Marek Skiba , Md. Nahiduzzaman Rose , moopet , msohng , Noah van der Aa , O. Jones , OverCoder , Panda , Parth Patel , rap-2-h , rhavendc , Romain Vincent , YCF_L
2	ALTER TABLE	e4c5 , JohnLBevan , kolunar , LiuYan , Matas Vaitkevicius , mayojava , Rick James , Steve Chambers , Thuta Aung , WAF , YCF_L
3	Arithmetic	Barranka , Dinidu , Drew , JonMark Perry , O. Jones , RamenChef , Richard Hamilton , Rick James
4	Backticks	Drew , SuperDJ
5	Backup using mysqldump	agold , Asaph , Barranka , Batsu , KalenGi , Mark Amery , Matthew , mnoronha , Ponnarasu , RamenChef , Rick James , still_learning , strangeqargo , Sumit Gupta , Timothy , WAF
6	Change Password	e4c5 , Hardik Kanjariya ツ , Rick James , Viktor , ydaetskcoR
7	Character Sets and Collations	frian , Rick , Rick James
8	Clustering	Drew , Rick James
9	Comment Mysql	Franck Dernoncourt , Rick James , WAF , YCF_L
10	Configuration and tuning	ChintaMoney , CodeWarrior , Epodax , Eugene , jan_kiran , Rick James
11	Connecting with UTF-8 Using Various Programming language.	Epodax , Rick James
12	Converting from MyISAM to InnoDB	Ponnarasu , Rick James , yukoff
13	Create New User	Aminadav , Batsu , Hardik Kanjariya ツ , josinalvo , Rick

		James , WAF
14	Creating databases	Daniel Käfer , Drew , Ponnarasu , R.K123 , Rick James , still_learning
15	Customize PS1	Eugene , Wenzhong
16	Data Types	Batsu , dakab , Drew , Dylan Vander Berg , e4c5 , juergen d , MohaMad , Richard Hamilton , Rick James
17	Date and Time Operations	Abhishek Aggrawal , Drew , Matt S , O. Jones , Rick James , Sumit Gupta
18	Dealing with sparse or missing data	Batsu , Nate Vaughan
19	DELETE	Batsu , Drew , e4c5 , ForguesR , gabe3886 , Khurram , Parth Patel , Ponnarasu , Rick James , strangeqargo , WAF , whrrgarbl , ypercube , Илья Плотников
20	Drop Table	Noah van der Aa , Parth Patel , Ponnarasu , R.K123 , Rick James , trf , Tushar patel , YCF_L
21	Dynamic Un-Pivot Table using Prepared Statement	rpd
22	ENUM	Philipp , Rick James
23	Error 1055: ONLY_FULL_GROUP_BY: something is not in GROUP BY clause ...	Damian Yerrick , O. Jones
24	Error codes	Drew , e4c5 , juergen d , Lucas Paolillo , O. Jones , Ponnarasu , Rick James , WAF , Wojciech Kazior
25	Events	Drew , rene
26	Extract values from JSON type	MohaMad
27	Full-Text search	O. Jones
28	Group By	Adam , Filipe Martins , Lijo , Rick James , Thuta Aung , WAF , whrrgarbl
29	Handling Time Zones	O. Jones
30	Indexes and Keys	Alex Recarey , Barranka , Ben Visness , Drew , kolunar , Rick James , Sanjeev kumar

31	INSERT	0x49D1 , AbcAeffchen , Abubakkar , Aukhan , CGritton , Dinidu , Dreamer , Drew , e4c5 , fnkr , gabe3886 , Horen , Hugo Buff , Ian Kenney , Johan , Magisch , NEER , Parth Patel , Philipp , Rick James , Riho , strangeqargo , Thuta Aung , zeppelin
32	Install Mysql container with Docker-Compose	Marc Alff , molavec
33	Joins	Artisan72 , Batsu , Benvorth , Bikash P , Drew , Matt , Philipp , Rick , Rick James , user3617558
34	JOINS: Join 3 table with the same name of id.	FMashiro
35	JSON	A. Raza , Ben , Drew , e4c5 , Manatax , Mark Amery , MohaMad , phatfingers , Rick James , sunkuet02
36	Limit and Offset	Alvaro Flaño Larrondo , Ani Menon , animuson , ChaoticTwist , Chris Rasys , CPHPython , Ian Gregory , Matt S , Rick James , Sumit Gupta , WAF
37	LOAD DATA INFILE	aries12 , Asaph , bhrached , CGritton , e4c5 , RamenChef , Rick James , WAF
38	Log files	Drew , Rick James
39	Many-to-many Mapping table	Rick James
40	MyISAM Engine	Rick James
41	MySQL Admin	Florian Genser , Matas Vaitkevicius , RationalDev , Rick James
42	MySQL client	Batsu , Nathaniel Ford , Rick James
43	MySQL LOCK TABLE	Ponnarasu , Rick James , vijeeshin
44	Mysql Performance Tips	arushi , RamenChef , Rick James , Rodrigo Darti da Costa
45	MySQL Unions	Ani Menon , Rick James
46	mysqlimport	Batsu
47	NULL	Rick James , Sumit Gupta
48	One to Many	falsefive
49	ORDER BY	Florian Genser , Rick James

50	Partitioning	Majid , Rick James
51	Performance Tuning	e4c5 , RamenChef , Rick James
52	Pivot queries	Barranka
53	PREPARE Statements	kolunar , Rick James , winter
54	Recover and reset the default root password for MySQL 5.7+	Lahiru , ParthaSen
55	Recover from lost root password	BacLuc , Jen R
56	Regular Expressions	user2314737 , YCF_L
57	Replication	Ponnarasu
58	Reserved Words	juergen d , user2314737
59	Security via GRANTS	Rick James
60	SELECT	Ani Menon , Asjad Athick , Benvorth , Bhavin Solanki , Chip , Drew , greatwolf , Inzimam Tariq IT , julienc , KartikKannapur , Kruti Patel , Matthis Kohli , O. Jones , Ponnarasu , Rick James , SeeuD1 , ThisIsImpossible , timmyRS , YCF_L , ypercube
61	Server Information	FMashiro
62	SSL Connection Setup	4444 , a coder , Eugene
63	Stored routines (procedures and functions)	Abhishek Aggrawal , Abubakkar , Darwin von Corax , Dinidu , Drew , e4c5 , juergen d , kolunar , Ilanato , Rick James , userlond
64	String operations	Abubakkar , Batsu , juergen d , kolunar , Rick James , uruloke , WAF
65	Table Creation	4444 , Alex Shesterov , alex9311 , andygeers , Aryo , Asaph , Barranka , Benvorth , Brad Larson , CPHPython , Darwin von Corax , Dinidu , Drew , fedorqui , HCarrasko , Jean Vitor , John M , Matt , Misa Lazovic , Panda , Parth Patel , Paulo Freitas , Přemysl Šťastný , Rick , Rick James , Ronnie Wang , Saroj Sasmal , Sebastian Brosch , skytreader , Stefan Rogin , Strawberry , Timothy , ultrajohn , user6655061 , vijaykumar , Vini.g.fer , Vladimir Kovpak , WAF , YCF_L , Yury Fedorov

66	Temporary Tables	Ponnarasu , Rick James
67	Time with subsecond precision	O. Jones
68	Transaction	Ponnarasu , Rick James
69	TRIGGERS	Blag , e4c5 , Matas Vaitkevicius , ratchet , WAF , YCF_L
70	UNION	Matthew Whitt , Rick James , Riho , Tarik , wangengzheng
71	UPDATE	4thfloorstudios , Chris , Drew , Khurram , Ponnarasu , Rick James , Sevle
72	Using Variables	kolunar , user6655061
73	VIEW	Abhishek Aggrawal , Divya , e4c5 , Marina K. , Nikita Kurtin , Ponnarasu , R.K123 , ratchet , Rick James , WAF , Yury Fedorov , Илья Плотников