MySQL Survival Guide

Based on Dalhousie's
CSCI-2141: Intro to Databases
Feb. 2024 – Aug. 2024

Gab 'Sp0k' Savard

This started as a personal MySQL help file, catered to my Ubuntu machine. I wrote it for Dalhousie's CSCI-2141: Intro to Databases taught by Dr. Beiko and Dr. Malloch. The guide does not cover theory, or at least not a lot of theory, as its only goal was to be a cheat sheet for coding MySQL.

Last updated: 2024-08-12
Up to: Week 5

Contents

Utilities

In this section, you will find the commands to start, stop and connect to your MySQL server. I recommend turning them into script saved on your machine that you can run to make your life easier. Each of them will be associated with my script.

Start the dbms

```
sudo systemctl start mysql
```

or

```
startMySQL # Personal Script
```

Stop the dbms

```
sudo systemctl stop mysql
```

or

```
stopMySQL # Personal Script
```

Connect to the server

```
sudo mysql -u root
```

or

```
mysql -u <username> -p
```

Where *-p* is only added if the user has a password set. A prompt will appear asking for the password.
or

```
connectMySQL # Personal Script
```

Read a file into a table (file being on your machine)

```
sudo mysql -u root < <file_name>
```

or

```
mysql -u root -p < <file_name>
```

or

```
readMySQL <file_name> # Personal script
```

* *All of these commands can be done through the workbench app as well.* *

Inside MySQL Server

In this section, you will find different commands to use once you are connected to your MySQL server.

SHOW

The equivalent of 'ls' in bash inside your server.

```
-- Shows all elements of the type precised
SHOW DATABASES;
SHOW TABLES;
```

* *Reminder: Commands and names of classes inside MySQL should be capitalized* *

USE

Opens the database specified. Any query executed will be executed for this DB.

```
-- Selects a database to work with
USE DatabaseDB;
```

* *Reminder: Usually, databases name will have a capitalized first letter and a DB at the end (i.e. DatabaseDB). Table names will be all lowercase.* *

Queries

SELECT

The `SELECT` query generates customized results to a query. It does **NOT** modify the data in the table!

```
-- A SELECT query with specific columns and styling
SELECT columns
  FROM table_name
  WHERE criteria
  ORDER BY column_name
  LIMIT max_rows;
```

```
SELECT * FROM table_name; -- -- Returns every columns from a table.
```

AS

Gives a display name to a column when printing the results of the query.

```
SELECT column1 AS 'name1', column2 AS 'name2', column3 AS 'name3', ...
  FROM table_name;
```

WHERE

Specifies one or more criteria that restricts the rows returned.

- Comparison operators: `=`, `>`, `<`, `>=`, `<=`, `<>`, `!=`, `BETWEEN`, `LIKE`
- Logical operators: `AND`, `OR`, `NOT`
- Set operator: `IN`
- Defined: `IS [NOT] NULL`

* *Matching is **NOT** case sensitive* *

Here is how to use each:

Comparison Operators
These operators mean the same thing as what they would mean in different coding languages. There is however specific ones to MySQL.

Symbols	Meaning
`=`	Equals to
`>`	Bigger than
`>=`	Bigger or equal to
`<`	Smaller than
`<=`	Smaller or equal to
`<>` / `!=`	Not equal to
`BETWEEN`	Declares a range for the value to be between
`LIKE`	Adds more flexibility to string matching

Example:

```
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column1 = value; -- This case looks for the elements that match the value.
```

Logical Operators
Allow more than one condition for the different attributes.

Symbols	Meaning
`AND`	Equivalent to `&&` in C, the condition is true if all conditions are respected
`OR`	Equivalent to `||` in C, the condition is true if at least one of the condition are respected
`NOT`	Equivalent to `!` in C, the condition is true if it is not respected

Example:

```
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE (column1 = 'value1' OR column1 <> 'value2') AND NOT column2 = 'value3';
```

Set Operators
The set operator `IN` (the only one seen in this guide) is a shorthand for multiple OR operator for the same attribute.

Example:

```
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column IN (value_1, value_2, ...);
```

LIKE
Like mentioned above, the `LIKE` operator adds more flexibility to string matching. It is still limited though, it can't easily match:

- Any string that has the same first and last letter ("Australia", "Asia")
- Specific character classes ("F" followed by a vowel requires 5 OR statements)
- Guaranteed case matching

- Return subsets of strings
- Requires regular expressions for this (Out of scope for this guide)

The `LIKE` operator uses two special characters to allow wildcard matches:

Character	Meaning
`%`	Match zero or more characters of any type
`_`	Match exactly one character of any type

It may or may not be case sensitive.

Examples:

Like with `%`:

```
-- Looks for everything that starts with "A" or "a"
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column LIKE "%A";

-- Looks for everything that does not contain "A" or "a"
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column NOT LIKE "%A%";
```

LIKE wildcard matching:

```
-- Looks for everything containing 'rain'
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column LIKE '%rain%';

-- Looks for everything starting by a character followed by 'rain'
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column LIKE '_rain';
```

** Each _ added means one extra character. **

Multiple LIKE conditions:

```
-- Looks for every string where at least one of them starts with 'A'
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column LIKE "A% %" -- First word
    OR column LIKE "% A%"; -- Non-first word
```

NOT LIKE:

```
-- Looks for every string who's second character is not "A"
SELECT column1, column2, column3, ...
  FROM table_name
  WHERE column NOT LIKE "_A%";
```

CREATE

The operations can be used directly in the MySQL server or written on a file along with the INSERT operations and read/ran into the server later.

Create a Database

There is two options to create new databases.
Option 1: Don't create a new one on top of an old one.

```
-- Ensures if the database already exist it won't be overwrittten
CREATE DATABASE IF NOT EXISTS NameDB;
```

Option 2: Ensure a new one is created (Will cause an error if the database does not already exist)

```
-- Erases the database and recreates it
DROP DATABASE NameDB;
CREATE DATABASE NameDB;
```

* *A database name usually has a first capitalized letter and ends with a capitalized DB (i.e. CitiesDB)* *

Both option of database creation can also be used together to ensure less error.

```
DROP DATABASE IF EXISTS NameDB;
CREATE DATABASE IF NOT EXISTS NameDB;
```

Create a Table

There is also two options to create new tables. The same logic can also be used to combine both options.
Option 1: Don't create a table with the same name as an existing one

```
CREATE TABLE IF NOT EXISTS table_name (...);
```

Option 2: Ensure a new fresh table is created

```
DROP TABLE table_name;
CREATE TABLE table_name;
```

When creating a new table, one should add parameters to the table

```
CREATE TABLE table_name (
  column_1 DATATYPE_CONSTRAINT(S),
  column_2 DATATYPE_CONSTRAINT(S),
  PRIMARY KEY (column_1),
  ...
);
```

Example of a table creation taken from Dr. Beiko's slides:

```
CREATE TABLE cities_simplified (
  city_id INT,
  city_name VARCHAR(100) NOT NULL,
  country_name VARCHAR(100) NOT NULL,
  year_founded INT,
  PRIMARY KEY (city_id),
  UNIQUE(country_name),
  UNIQUE(city_name, year_founded)
);
```

Datatypes

Datatype	Definition
`VARCHAR(n)`	Variable-length character string (n <= 65535)
`TEXT`	Unstructured text. Can be much longer, but very slow
`INT`	Integer (signed by default), signed range is (-2147483647, 2147483647)
`FLOAT` / `DOUBLE`	32-bit/64-bit floating point
`DECIMAL(x,y)`	x represent the total digits, y represent the number of digits after the decimal point. (ex: DECIMAL(314, 2) = 3.14)
`DATE`	Holds a string in the format yyyy-mm-dd
`TIME`	Saves a time value in the format hh:mm:ss[.fraction] -> The time datatype has a lot of prebuilt function like the TIMEDIFF that will calculate the difference between two time values.

Constraints

Define limitations on columns. You cannot insert into a table that violates those constraints.

Constraint	Definition
`NOT NULL`	Column must be defined for every entity that is added
`UNIQUE`	Duplicate values cannot be added
`DEFAULT`	If no value is specified, set the value to this
`CHECK`	Allow entity to be added only if column of table values satisfy constraints
`AUTO_INCREMENT`	Automatically give the numerical value if it is not specified while incrementing the count

Examples:
These examples are taken directly from Dr. Beiko's slides.

Create table using DEFAULT:

```
CREATE TABLE cities_simplified(
  city_id INT,
  city_name VARCHAR(100) NOT NULL,
  country_name VARCHAR(100) NOT NULL,
  year_founded INT DEFAULT 2024,
  PRIMARY KEY(city_id),
  UNIQUE(country_name),
  UNIQUE(city_name, year_founded)
);
```

Create table using CHECK constraints:

```
CREATE TABLE cities_simplified(
  city_id INT,
  city_name VARCHAR(100) NOT NULL,
  country_name VARCHAR NOT NULL,
  year_founded INT
    CONSTRAINTS 'year_range'
    CHECK(year_founded BETWEEN -5000 and 2024),
  PRIMARY KEY(city_id),
  UNIQUE(country_name),
  UNIQUE(city_name, year_founded)
);
```

Create table with AUTO_INCREMENT:

```
CREATE TABLE cities_simplified(
  city_id INT AUTO_INCREMENT,
  city_name VARCHAR(100) NOT NULL,
  country_name VARCHAR(100) NOT NULL,
  year_founded INT,
  PRIMARY KEY(city_id),
  UNIQUE(country_name),
  UNIQUE(city_name, year_founded)
);
```

ALTER

The ALTER query can be used to modified the data already saved in the tables.

* *Be careful, sometimes, modification will be rejected if it contradicts the data already in the database.* *

Examples:
Add a column:

```
ALTER TABLE table_name
  ADD (new_colname DATATYPE constraints);
```

Drop a column:

```
ALTER TABLE table_name
  DROP COLUMN column_name;
```

Modify column type:

```
ALTER TABLE table_name
  MODIFY COLUMN column_name new_datatype;
```

Change column name:

```
ALTER TABLE table_name
  CHANGE old_column_name new_column_name;
```

Add primary key:

```
ALTER TABLE table_name
  ADD PRIMARY KEY (column_name);
```

Drop primary key:

```
ALTER TABLE table_name
  DROP PRIMARY KEY;
```

Add foreign key:

```
ALTER TABLE table_name
  ADD CONSTRAINT fk_name
  FOREIGN KEY (column_name)
  REFERENCES parent_table(parent_column_name);
```

Drop foreign key:

```
ALTER TABLE table_name
  DROP FOREIGN KEY fk_name;
```

Add index:

```
ALTER TABLE table_name
  ADD INDEX index_name (column_name);
```

Drop index:

```
ALTER TABLE table_name
  DROP INDEX index_name;
```

Rename table:

```
ALTER TABLE old_table_name
  RENAME TO new_table_name;
```

Add unique constraint:

```
ALTER TABLE table_name
  ADD UNIQUE (column_name);
```

Drop unique constraint:

```
ALTER TABLE table_name
  DROP INDEX index_name;
```

DROP

Deletes the specified table or database.

Database:

```
DROP DATABASE Database_nameDB;
```

Table:

```
DROP TABLE table_name;
```

INSERT

The INSERT statement is used to add new entities(rows) to a table.

```
INSERT INTO table_name(column_name1, column_name2, column_name3, ...)
VALUES (value1, value2, value3, ...);
```

or

```
INSERT INTO table_name(column_name1, column_name2, column_name3, ...)
VALUES
(value1, value2, value3, ...),
(value1, value2, value3, ...),
(value1, value2, value3, ...),
```

```
...
(value1, value2, value3, ...);
```

One can also use the IGNORE keyword in order to force the entry of the data even if it violates the constraints. IGNORE will only result in a warning. But this must be used with precaution.

UPDATE

The UPDATE statement is used to update a table with new values for columns and entities.

```
UPDATE table_name
  SET column1 = value1, column2 = value2, column3 = value3, ...
  WHERE conditions;
```

DELETE

The DELETE statement is used to delete a row in a table.

```
DELETE FROM table_name
  WHERE conditions;
```

Derived Attributes

In addition to returning values contained in the table, one can combine values from different columns using various operators.

- Mathematical operators: `+`, `-`, `*`, `/`
- Text operators: `CONCAT()`

The idea is to be able to use them to mix values, for example calculating the population density:

```
SELECT population/land_area
  FROM table_name
  WHERE condition;
```

Rounding

`ROUND(attribute, digits)`: Round attribute to digits of precision (default is zero).

```
SELECT ROUND(1.1111); -- Will be equal to 1
SELECT ROUND(1.1111, 2); -- Will be equal to 1.11
```

The `ROUND` command can renamed with an alias using AS.

```
SELECT column1, column2, column3, ROUND(column3/column2, 1) AS alias
  FROM table_name;
```

* *Reminder that the WHERE clause does not recognize aliases.* *

Aggregation

Aggregation can be summarized as “give me a row that summarizes one or more rows in the table”. In order to perform an aggregation, one must use aggregation operators.

Operator	Definition
`COUNT`	Total rows satisfying the criteria
`MAX`, `MIN`	Max and min values of a given column for all rows satisfying the criteria
`SUM`, `AVG`	Sum and average of values in a given column for all rows satisfying the criteria

COUNT

How many rows satisfy a given condition?

```
SELECT COUNT(*) FROM table_name;
```

```
SELECT COUNT(column) FROM table_name;
```

```
SELECT COUNT(column)
  FROM table_name
  WHERE condition;
```

DISTINCT

Shows each of the distinct values once, even if they appear more than once in the table.

Display distinct values:

```
SELECT DISTINCT column
  FROM table_name;
```

Display the number of distinct values

```
SELECT COUNT(DISTINCT(column)) AS 'name'
  FROM table_name;
```

MAX, MIN, SUM, AVG

With numbers:

Identify the average value of an attribute:

```
SELECT AVG(column) AS 'name'
  FROM table_name;
```

Identify the max value of an attribute:

```
SELECT MAX(column) FROM table_name;
```

Identify the min value of an attribute:

```
SELECT MIN(column) FROM table_name;
```

Identify the sum of the value of an attribute:

```
SELECT SUM(column) FROM table_name;
```

With strings, sort of…
MAX returns the last value in alphabetical order:

```
SELECT MAX(column) FROM table_name;
```

MIN returns the first value in alphabetical order:

```
SELECT MIN(column) FROM table_name;
```

AVG returns nothing, the average of strings has no meaning.

SUM does not apply to strings.

Connecting Tables

Connecting tables depends on the existence of attributes that are shared between tables. Common attributes can exists across different tables. Foreign keys can be used to make the relationship between those tables explicit, and impose important constraints.

NOTE: The foreign key attribute does not need to have the same name as the primary-key attribute in the parent table!

Foreign Keys

A foreign key is a primary key from a parent table placed into a dependent table to create a common attribute. The foreign key must be unique. The foreign key must be constrained to ensure referential integrity. Furthermore, the referenced attribute can be any candidate key (including composite key).

A referential integrity is a condition by which a dependent table's foreign key entry must have either a null entry, or a matching entry in the primary key or the related table.

Creating foreign keys

The parent table must already exist before the foreign-key relationship is created!
A foreign key can be added during the creation of a new table.

```
CREATE TABLE IF NOT EXISTS table_2 (
  column_1 DATATYPE constraint UNIQUE,
  column_2 DATATYPE constraint,
  column_3 DATATYPE constraint,
  ...
  PRIMARY KEY(column_1),
  FOREIGN KEY(column_2) REFERENCES table_1(column_ref)
);
```

Reminder on how to add a foreign key to an already existing table:

```
ALTER TABLE table_name
  ADD FOREIGN KEY (column_name)
  REFERENCES parent_table(parent_column_name);
```

Table Joins

So far, this guide has been over how to interact with one table at a time. But sometimes, like mentioned earlier, one could want to retrieve information that is stored across different tables. For example: The information of a developer of a specific video game in a table. Table joins combine data from two or more tables to identify associations between entities.

For now, this guide has shown the following code:

```
SELECT column_names
  FROM table_name
  WHERE conditions
  GROUP BY criterion
  HAVING criterion
  ORDER BY criterion
  LIMIT row_count;
```

This lets a user get all the information they would want from one table. But, MySQL lets them also request information from different tables in a single statement. The code would now look like this:

```
SELECT column_names
  FROM table_1 JOIN table_2
    ON (column_name1 = column_name2)/USING(colname) -- USING is simpler to use, in my opinion
  WHERE conditions
  GROUP BY criterion
  HAVING criterion
  ORDER BY criterion
  LIMIT row_count;
```

Referencing Tables

Selecting columns when there is multiple tables in the statement looks a little like using objects when coding with Java. As one write their query, they have to specify which table to get the information from using either the name of the table or an alias.

```
SELECT t1.column_1, t1.column_2, t1.column_3
  FROM table_1 t1;
```

Using aliases or table names like the example above is unnecessary when there is only one table, but essential the moment the statement requires two or more tables.

```
SELECT t1.column_1, t1.column_2, t2.column_a
  FROM table_1 t1 JOIN table_2 t2 USING(column_a);
```

In the example above, the column_a would require to be the foreign key used to connect the two tables.

What about multiple foreign keys?

A table can have multiple keys pointing to the same key, it just means that at least one of the keys' name will not be the same as the parent table's key it is referencing.

JOINS

Before continuing with the code, I think it is important to go over a little theory about joins:

Type of Join	Definition
`CROSS JOIN`	Join every row in table 1 with every row in table 2
`INNER JOIN`	Join two tables on some column, returning no results for non-matching rows
`NATURAL JOIN`	Join two tables using every shared column name
`(LEFT, RIGHT) OUTER JOIN`	Join two tables, potentially including some unmatched rows

CROSS JOIN

The CROSS JOIN, sometimes referred to as the most basic join, combines every row of Table 1 with every row of Table 2. It yields the Cartesian product, which is usually not very useful on its own.

```
SELECT * FROM table_name1 CROSS JOIN table_name2;
```

There also exists a different way to do the `CROSS JOIN` operation using an older syntax. It consists of replacing the `CROSS JOIN` terms themselves by a `,`. This makes the code more compact but also less explicit, which, when writing databases with multiple programmer, would be bad.

```
SELECT * FROM table_name1, table_name2;
```

To give an example of how the math would work, if one was to use a CROSS JOIN on a table of 4 rows and 6 columns and another table of 4 rows and 5 columns, then they would find themselves with a combined table of 16 rows and 11 columns.

Using the Cartesian product is not always a good idea though, as it would match informations in ways that don't necessarily make sense. Like matching an information on an entity that isn't stored in that specific row for example. In order to solve this problem, one would have to establish a restriction using the WHERE keyword, for example:

```
SELECT *
  FROM table_name1 t1 CROSS JOIN table_name2 t2
  WHERE t1.column_1 = t2.column_A;
```

INNER JOIN

An INNER JOIN is the equivalent of a CROSS JOIN (everything * everything), but with a restriction to the matching criteria.

Just like the CROSS JOIN, the INNER JOIN also has an alternative form: JOIN. The four following commands would all yield the same result:

```
SELECT * FROM table_name1, table_name2; -- Implicit terminology, restricts with WHERE
```

```
SELECT * FROM table_name1 CROSS JOIN table_name2; -- Explicit terminology, restricts with WHERE
```

```
SELECT * FROM table_name1 JOIN table_name2; -- Implicit terminology, restricts with ON/USING
```

```
SELECT * FROM table_name1 INNER JOIN table_name2; -- Explicit terminology, restricts with ON/USING
```

ON and USING Operators

JOIN…ON: Identify specific column names from each table to use for joining (they **do not** need to have the same names).

JOIN…USING: Identify shared column names to use for joining (they **do** need to have the same names).

ON is good because the joining columns don't need to have the same name. But both columns are returned in the result, which can be confusing/redundant.

USING is good because it returns only one of the matching columns, but both must have the same name.

NATURAL JOIN

The NATURAL JOIN will assume that the user wants to join all attributes with shared names between the two tables. Meaning, if two table have the same information, it will not repeat it. In case there is no common column, the NATURAL JOIN will yield a Cartesian product. The same will happen if datatypes don't match.

As great as the NATURAL JOIN is, Dr. Beiko and Dr. Malloch don't recommend its use. It saves some complexity, but is completely sensitive to the database design, it makes assumptions and has a weird unintuitive behavior if nothing is shared.

OUTER JOIN

An OUTER JOIN is the equivalent of a CROSS JOIN (everything * everything), restricted to matching criteria, plus non-matching rows.

Compared to the INNER JOIN, the OUTER JOIN will also include lines with no corresponding values. Or in simpler words: OUTER JOIN returns rows that match from both table and rows from one table that have no matches in the other.

The OUTER JOIN also has "add-ons" for the command to specify which table to take the non-matching rows from:

Join	Definition
`LEFT OUTER JOIN`	Every row in table 1 that doesn't match table 2
`RIGHT OUTER JOIN`	Every row in table 2 that doesn't match table 1

Here is how one would write each of the commands for the different specifications:

```
SELECT *
  FROM table_name1 t1
  LEFT OUTER JOIN table_name2 t2
  ON t1.column_1 = t2.column_a;
```

```
SELECT *
  FROM table_name1 t1
  RIGHT OUTER JOIN table_name2 t2
  ON t1.column_1 = t2.column_a;
```

SELF JOIN ?

A self join is not a different class of JOIN. It can be performed with any of the JOINS seen above. The syntax and function are exactly the same, but it lets you explore very interesting connections in the data.