# SQL (MYSQL) Notes EAYL week 1

# Abstract theory

Applications have a need to store data, relational databases have become the de-facto way to store enterprise for numerous reasons.

A relational database is a group of tables, each one containing information about a certain kind of thing

**Relational theory**

**Attribute:** The term used to describe a named column or field within a row.

**Tuple:** This term is what is normally called a row and is formally described as an ordered set of attributes.

**Relation:** This Is what we would call a table and is a set of tuples or rows.

**Tables, Columns and Rows**

Within the context of relational databases, tables are made up of columns and rows. They may appear like spreadsheets or webpage tables but there are significant differences.

Spreadsheets are designed to be used within a single application and support flexible analytics within a single sheet. There is no real distinction between columns and rows and they may be interchanged without losing any meaning, furthermore individual cells can contain literal values or expressions to calculate values based on the values from other cells.

Tables in a relational database are much more rigidly defined. Each column has a *name* and a *data type* and *values within those columns* must conform to the specified data type. Calculated values are supported by many database platforms, but they must apply to every value in that column, you cannot pick and choose.

**Tables (cont.)**

Each **table** contains information about a certain kind of thing, the **columns** define the nature of the information we store about each thing and each **row** is one of the *things* we’re storing.

A table typically has much fewer columns than rows (rare that within a table, there are more attributes about an object that we store than number of actual objects)

Databases are often shared between many different client programs, For example in a full stack e-commerce situation (think Amazon) The same database could be used to track your order from the **point of purchase**, through to **warehouse retrieval** and **delivery instructions** right through to after-purchase **business analytics** . All of which using different platforms to access the same data within the same database.

**Relational Database Management Systems**

A Relational Database Management System (**RDBMS)** is a server program shared between multiple client programs. They typically make use of the client/server paradigm. (where a client program requests some service of a server program, often running on a separate physical machine).

The client server model allows separation of concerns where the server program manages data, constrains and concurrency issues such as locking and transaction handling and persistence. This allows the client programs to focus on processing and presenting the data to users or other systems (instead of having to worry about handling all expensive database stuff itself).

Larger applications make use of multi-tier architectures, where the RDBMS becomes part of the *data tier.* The remaining portions of the application are held in different tiers. E.g. the MVC framework (Model, view controller). Where each tier may be split into functional layers handling different features. In MVC the data behind the objects in the model are often persisted in an RDBMS.

*Note: Database platforms such as Microsoft Access or SQLite are not RDBMS platforms but are rather file-based systems where the database is manifested by a file rather than a server program and accessed locally by an application on the same host as the file.*

**Popular Database:**

There are many popular RDBMS platforms used in various niches, from large clusters running at enterprise level to a single instance hosting a small website or project.

**Oracle:** The oldest and most widely used database platform in enterprise applications

**MySQL:** The most widely used open-source database, widely used on the web and in embedded applications

**SQL Server:** Microsoft’s commercial database platform, has gained much traction in the enterprise since SQL Server 2000.

**PostgreSQL:** An open-source database with no corporate owner and an emphasis on standards compliance and advanced features.

**NoSQL Data Stores**

Not all data is relational, sometimes a big lookup list of keys and values for fast memory-resident retrieval across a cluster of systems in a web environment. NoSQL is a term used to describe data stores that do not (only) use SQL to store data. The actual method of storage differs across systems; some use *Key/value stores* whereas others store entire documents in the form of JSON documents. In common they have a simplified API that does not use SQL and so avoids the overhead of parsing, standardizing and optimizing statements in that language. As such NoSQL as a term is more useful by what it excludes rather than what it includes. Most used NoSQL: MongoDB, Memcached, Redis, Cassandra, Couchbase.

# MySQL

Steps to open

* Sudo bash
* Mysql

Show all databases in the system

* SHOW DATABASES;

A single instance of MySQL can host multiple databases, this command shows all databases in the current instance of MySQL.

## Creating a database and a simple table

* CREATE DATABASE demo;
* SHOW DATABASES;
* USE demo;
* CREATE TABLE simple\_table(  
  id INT PRIMARY KEY,  
  name VARCHAR(20)  
  );
* INSERT INTO simple\_table (id,name)  
  VALUES(1,’Bob’);

*The Insert command maps a value to each column specified*

Select the entire contents of the table with the following statement

* SELECT \* FROM simple\_table;

*The \* means all columns, e.g. Name and id in this case*

## Updating/editing a row within a table

* Update simple\_table
* SET name = ‘Pete’
* Where id=1;

## Deleting a row within a table

* Delete From simple\_table
* Where id = 1;

## Creating and writing scripts

SQL scripts are incredibly powerful as it allows configurations to be easily recreated on multiple machines without the need to actually transfer any data, only the code itself used to create the database.

### Creating and importing from a script

*To be written in a text editor*

* */\* Test script \*/*
* *Create Database test\_from\_script;*
* *Use test\_from\_script*
* *Create table test\_table(I int, v varchar(20);*
* *Insert into test\_table values(1,’hello’),(2,’world’);*

*In the mysql client, execute the following*

* Source testscript.sql

Source command executes all statements in the file identified by it’s parameter

*Note: error “2” signifies file/path not found when source is called.*

### Efficient database import script

* Create database world;
* Use world**;**
* set autocommit=0;
* source xyz/xyz.sql
* source abc/123.sq;
* set autocommit=1;

The auto commit variables specifies whether each statement is treated as a single transaction. Disabling it allows an entire sequence of statements to be processed as one transaction which speeds up the script execution.

## Viewing Metadata

Using the show command to display metadata for tables and other objects like indexes.

## Display all tables

* Show databases;
* Use world;
* Show tables;

*Alternatively, to see a more specific list, use*

* Show full tables;

*To see even more in-depth information use*

* Show tables status\G

This command shows the storage engine, along with estimated row length and data size used and other useful information about each table

*The \G terminator is similar to semicolon but presents columns vertically, if you try this statement with a semicolon instead of \G you will see that the output is easier to read with vertical output.*

## The Describe command

* “Describe world.city”

The most common command used to view table metadata is DESC, which is a shortened form of DESCRIBE, DESCRIBE is a synonym for SHOW CIOLUMNS FROM. These commands show the columns defined in a table, along with their data types, index information and other metadata

The output of a DESCRIBE command is in itself in the form of a table

The output contains the following columns

* **Field** The column name of the table being described
* **Type** The data type of the column
* **Null** Whether the column allows Null values
* **Key** Type of index the column participates in, if any
* **Default** The default value of the column
* **Extra** Additional information including whether the column is auto-incrementing or an auto-updating timestamp.
* **Collation** The collation used for character string columns
* **Privileges** Privileges that the user executing the show command has for the column
* **Comment** Comments for the column, as specified by the CREATE TABLE command used to create the table

## Viewing Table Definitions

Execute the following commands to browse to and then describe the structure of the city table.

* Use world;
* Show tables;
* Desc city;

Use the **explain** command to describe the structure of the country table

* Explain country;

Display even more information about the table by issuing the following command

* Show full columns from country;

Display the statement used to create the table with the following statement

* Show create table country\G

*This is a really useful command for trying to find out how to recreate a table*

## Creating tables

To create tables yourself you must use **Create Table** statements similar to those you have seen when using pre-built scripts and the various metadata commands

### Creating tables

* Create a database called Orders and use it
* Create a table within Orders called Customers with the following command
  + Create table Customers(
  + ID int,
  + Name varchar(30),
  + Address varchar(100),
  + Phone char(64)
  + );
* Add the ID as a primary key
  + Alter table customers
  + Add Primary Key(ID);
* Set the ID field to auto-incrementing
  + **Alter** **Table** Customers **modify** ID int **AUTO\_INCREMENT**

*Now when a new record is inserted without an ID parameter specified, the database will automatically lookup the highest value of ID that is occupied, add 1 to it and set that as the ID of the record you’re inserting*

* Insert a row into the customer table
  + Insert into customers(ID,Name) Values (1,’First’);
* Insert another row into the customer table
  + Insert into customers(Name, Address,phone) values(‘second’,’abc 123’,’11223344’)

Notice that in the first insert statement we specified an ID and in the second we didn’t, both commands were accepted with no errors.

This means that when we specify a value for an auto-incrementing item of data, as long as it isn’t currently occupied it will be accepted, otherwise if the auto-incrementing value you specified already exists, it will throw a duplicate error.

In the second case where we didn’t supply an ID, the database automatically supplies the next available value for the newly inserted record to use.

**Using Datetime/Timestamps**

* **Create a table called Orders**
  + Create table Orders(
  + ID int primary key Auto\_incremement,
  + CustomerID int,
  + DispatchDate DATETIME
  + );

*The datetime type contains values that identify a point in time on a particular date*

* **Add a column called DatePlaced of type Timestamp by issuing the following command**
  + Alter table Orders
  + Add Column DatePlaced TimeStamp
  + After CustomerID;

***Note that the new column automatically sets the value of DatePlaced every time the table is updated, This is a feature of timestamp columns with the default settings.***

* **Change the definition of DatePlaced so that it updates automatically only when the row is first created**
  + Alter table orders
  + Modify column DatePlaced Timestamop
  + Default CURRENT\_TIMESTAMP

**Note: Setting a default value but not using the ON UPDATE CURRENT\_TIMESTAMP**  Syntax make the column value update automatically only when the row is first created.

**Dropping tables and Databases**

* Remove the test\_table form the test\_from)script database
  + Use test\_from\_script;
  + Drop table test\_table;

This command removes the table from the database

* Remove the test\_from\_script database
  + Drop database test\_from\_script

|  |  |  |
| --- | --- | --- |
| **Group** | **Data Type** | **Description** |
| Exact Numeric |  |  |
| Whole Number | Tinyint | 1 byte unsigned integer number |
|  | Smallint | 2 byte signed integer number |
|  | Mediumint | 3 byte signed integer number |
|  | Int | 4 byte signed integer number, also known as integer |
|  | Big int | 8 byte signed integer number |
|  | Bit | 1 bit – used to store bit field field values |
| Fixed point | Decimal[m,s] | Fixed precision (m) and scale (s). Default (m,s) is (10,0) |
|  | Numeric[m,s] | ANSI SQL synonym for decimal |
| Approx Numeric | Float | 4 byte floating point values |
|  | Double | 8 byte floating point values |
| Date and time | Date | Retrieved and displayed in ‘YYYY-MM-DD’ format, range from ‘1000-01-01’ to ‘9999-12-31’ |
|  | Datetime | Retrieved and displayed in ‘YYYY-MM-DD HH:MM:SS’ format, range from ‘1000-01-01 00:00:00’ to ‘9999-12-31 23:59:59’ |
|  | Timestamp | The same format as datetime but stored in (and values in the local timezone converted from and to) UTC in the range ‘1970-0101 00:00:01’ UTC to ‘2038-01-19 03:14:07’ |
|  | Time | Retrieved and displayed in ‘[H]HH:MM:SS’ fromat from ‘-838:59:59’ to ‘838:59:59’ |
|  | Year | Retrived and displayed in YYYY format, with a range of 1901 to 2155 or 0000 Fixed-width character struing with size of up to 255 characters |
| Character | Char(n) | Fixed-width character string with size of up to 255 chradcters |
|  | Varchar(n) | Variable length character string with maximum size up to 65,535 characters |
|  | Tinytext | Variable length character string of up to 255 (2^8 -1) |
|  | Text | Variable length character string up to 65,535 (2^16 -1) characters |
|  | Mediumtext | Variable length character string up to (2^24 -1) characters |
|  | Longtext | Variable length character string up to (2^32 -1) characters |
| Binary | Binary(n) | Must have the same fixed-length (up to 8KB for each row) |
|  | Varbinary(n) | Entries can vary in number of hexadecimal digits they contain (up to 8KB) |
|  | Tinyblob | Variable length binary data up to 255 bytes in length |
|  | Blob | Variable length binary data up to 65,535 bytes in length |
|  | Medium blob | Variable length binary data up to 2^24-1 bytes in length |
|  | Long blob | Variable length binary data up to 2^32 -1 bytes in length |
| List types | Enum | String value chosen from a fixed list defined in the column |
|  | Set | Zero or more string values chosen from a list defined in the column |

**Summary**

* Tables created using Create table statement
* Scripts are created as part of a database backup strategy or during software development as part of the version control strategy
* Tables are defined int terms of their colunns, each of which must have at least a name and a data type
* Further qualifies can be defined about each column such as primary/foreign key status, If it’s nullable or not and what default values are
* Different data types are available to store numeric data with different levels of precision
* The Alter Table statement allows you to modify the structure of a table that already exists
* You can drop tables using the Drop table statement

*(continue from pg 17 onwards)*

# Data integrity

## Data Integrity

When a table with columns is created, each column is given a data type e.g. varchar(n), in some cases such as char and varchar we specified a maximum length.

By setting a primary key on a table you specify that the values in the primary key’s column must be unique. In doing this you specified what type of data can be added to the columns in the table and therefore providing some kind of data integrity.

**Domain Integrity:** Ensuring values fall within appropriate ranges or sets

**Entity Integrity:** Ensuring each record references a unique *thing*

**Referential Integrity:** Ensuring entities are related to each other in suitable ways

## Domain Integrity

Data types provide one means of ensuring values are appropriate, i.e. you cannot store a value before 1970 in a column defined as timestamp, furthermore you can also use progrmattic means to perform further range checking in a number of ways:

* In your application code
* By creating stored routines that perform checks and requiring that all data modifications use those routines e.g. no direct access to the table (*Think views)*
* By using triggers that fire automatically every time an update occurs

Ideally all data modifications execute via such code. You cannot guarantee that all data modifications go through your application code, but you can use permissions to restrict access to the table and require the use of stored routines or use triggers that fire on every update.

The trade-off is in maintainability and complexity of the application as a whole when you consider all tiers presentation and data.

As an example, consider you have a database containing Irish car registration numbers. The numbers stored may look something like this

* 11D21509
* 151 MN 293
* 09-SO-8388

In each case, ignoring space and dash characters.

* First two or three characters represent the two-year digit year of the vehicle’s first registration which can be over 100 years ago
* In more recent years a 1 or a 2 representing the first or second half of the year of registration.
* The following one or two characters represent the registration district e.g. “D” for Dublin, “SO” for Sligo etc.
* Remaining characters are the sequence number for that district in that time period

You could enforce domain integrity for a column containing such registration numbers by creating a trigger to parse the string into portions based on delimiters (space or dash characters) and field widths

Then compare these portions with a list of registration districts (using FIND\_IN\_SET() or a subquery)

Simpler examples could include age restrictions on membership of a club or customer base, regular expression checks on string like email addresses or ensuring that an order is dispatched after it is placed.

## Entity Integrity

Uniqueness is enforced by primary keys and unique keys.

While a table can have **at most one primary key**, it can have several unique keys.

Each one defines one or more columns that taken together, must be unique within the table.

For example, you may have an employee’s table with an employee primary key, but each employee also has a unique email address that is not the primary key, the national insurance number must also be unique.

A database that allows for multiple concurrent employee to have the same email address or national insurance number is broken.

**Creating a unique key**

Create a unique index on the Products table from before with the following statement:

* **Create** **unique** **index** uniq\_prod\_name **On** Products(Name);

This statement creates a unique key(index) on the Name field of the Products table, ensuring that no two values are the same

Display the indexes on the products table by issuing the following statement:

* **Show** INDEXES **FROM** Products\G

You should see two indexes, one named *Primary*, the other created in the preceding step.

## Referential Integrity

**Relationships**

Defining and implementing relationships correctly is an essential part of implementing a database. There are three types of logical relationship:

**One-to-one:**

A record in one table has zero or one linked records in another table.

For example, a book may have optional promotional material- front cover image, promotional copy – stored in another table, but not all books may have this information.

**One-to-many:**

A record in one table has zero or more records in another table. For example, a publisher may have many related books, but each book has one (or zero) publishers.

This is the most common relationship type; other relationships are expressed in terms of this one

**Many-to-Many:**

A record in one table may have zero or more related records in the other table. The other table may have similarly many related records in the other.  
For example, a book may have many authors and each author may have many books.

This relationship is implemented by creating two one-to-many relationships with a special junction (*or linking)* table to represent the relationship. E.g. a “credits” table. *Each author may have several credits. Each book may have several credits; however each author may only have one credit per book and each book only one credit per author.*

## Foreign keys

A *foreign key* is a column that contains values referring to another column. When you create a foreign key constraint, you’re telling MySQL that values in the foreign key column are constrained so that they can only contain values that exist in the referenced columns.

*For example, in the pubs db.* If you look at the titles and publishers table, you can see that the pub\_id fields exist in both. The definition for the titles table includes the following definition.

* Create table ‘titles’ (
* ‘title\_id’ varchar(6) Not Null,
* …..
* Primary key (‘title\_id’),
* Key ‘pub\_id’ (‘pub\_id),
* Constraint ‘titles\_ibfk\_1’
* Foreign key (‘pub\_id’) references ‘publishers’ (‘pub\_id’)
* )

The **Constraint** configures a foreign key so that values in the pub\_id field must exist in the corresponding field in the publishers table.

Foreign key constraints exist within the full CRUD model.

e.g:

* Cannot create without referential integrity
* (obvs cant read if it cant be created)
* Cannot update to an invalid foreign key
* Cannot delete if object has foreign key dependency in other table.

For delete, if I tried to delete “Ireland” from the worlds database, it wouldn’t work, as in the “City” table, cities depend on Ireland to exist for referential integrity, if we deleted Ireland then these cities would have no country listed which is an invalid configuration.

In order to see why the “delete irl” command does not work

* **SHOW CREATE Table** City;

This shows the foreign key constraint enforces that values in the city.countrycode field must exist the country.code field.

This prevents new city rows that contain a CountryCode that does not exist, but also prevents deleting parent rows if the same condition could arise.

The relationship between city and country tables is a one-to-many relationship. Each city belongs to at most one country, but a country could have many cities. The one to many relationship is the most common type.

Each relationship is implemented byn at least one foreign key:

* One-To-Many relationships are implemented by a foreign key in one table linked to an identifying value in another
* One-To-One relationships have a unique or primary key constraint on the foreign key.
* Many-to-Many relationships consist of two one-to-many relationships with an intermediate junction table representing the relationship.

In reality databases tend to have many tables related in all interesting ways. Relationships can be very easily inferred and tested within MySQL.

## Creating and testing foreign key constraints

Use the following statements to create a foreign key constraint and therefore a relationship between the Customers and Orders tables

* Use orders;
* Alter table orders
* Add constraint customer\_exists
* Foreign Key(CustomerID)
* References Customers(ID);

*Note that no reference has to be made within the customer table*

## Summary of data integrity

* Data integrity is the principle where data means something more than just a set of values of a given data type
* There are different types of data integrity
  + Domain integrity
  + Entity integrity
  + Referential integrity
* Integrity can be enforced in application code or by using constraints
* Primary key and unique key constraints enforce uniqueness
* Referential integrity is enforced through foreign key constraints which in turn implement relationships between tables
* Relationships connect logically related tables, there are:
  + One-to-One relationships
  + One-to-many relationships
  + Many-to-many relationships
* Ultimately these are implemented as one or more, one-to-many relationships through the use of foreign keys
* Foreign keys normally restrict operations that change key values in referenced records but you can change that behavior by specifying a non-default referential action such as ON DELETE CASCADE

# Data Manipulation Language (DML)

## Overview

**Data Manipulation Language** (DML) is the subset of SQL concerned with working with data rather than the structure of the data. It consists of:

* The **SELECT** statement for querying
* **INSERT, REPLACE, UPDATE** and **DELETE.** For modifying data

When working with data you have access to lots of different syntactical features that make it easy to specify and query data, it’s rows and columns and process the values that you get back.

We’ll start with the most commonly used statement.

## The SELECT Statement

The SELECT statement is used to retrieve rows of data from a table or tables joined by a relationship. It specifies the anmes of the columns (or fields) you want to retrieve. FROM and WHERE are statements associated with select.

From allows you to specify what table oir tavles from which you wish to retrieve columns and the where clause identifies which rows in the table(s) you wish to retieve.

**General form**

The general form of the select statement is:

* **Select** select-list
* **From** table-list
* **Where** search-condition

For example the following code returns a list of first names and last names from the authors tbales where the field onctract has a value of 1 (for true)

* **Select** au\_fname, au\_lname
* **From** authors
* **Where** contract = 1;

Select is used to retrieve rows of information from one or more tables using selections, retrieve columns of information in projections and you can retrieve rows and columns from two or more tables in joins.

## Select \*

Select \* can be used to select all columns from the tables in the table list. For example, the following select statement lists all fields from the authors table who live in Oakland:

* Select \*
* From authors
* Where city = ‘Oakland’;

*Note that you should avoid using ‘\*’ in your code for the following reasons*

* The SQL statements and code used to process the data depend on the order of the columns when you run the query, not at the time you wrote the query
* You might return more data than you need and this adds extra load to the server, network and application

## Aliasing Column names in the SELECT statement

Columns in a result set can be re-named e.g.

* Select pub\_name as Publisher, pub\_id from Publishers;

By using quoted strings, an aliased column name can contain otherwise illegal characters such as spaces:

* Select pub\_name as ‘Book Publisher’ From Publishers;

Aliasing columns is useful when SQL reserved words are to be used for a re-named coliumn e.g.

* Select SUM(ytd\_sales\_ as ‘sum’ from titles;

Computations involving more than one column can be used:

* Select title, ytd\_sales \* price from titles;

**Operator Precedence:**

*Just note that MySQL will follow BIDMAS/BODMAS when deciding what order to execute commands in.*

**String expressions in SELECT clauses**

You can concatenate strings in expressions and column values with the CONCAT() function, e.g.:

* Select Concat(fname,’ ‘,lname) ‘Full Name’, ‘Newly hirded’, hire\_date from employee;

**Using Distinct – Eliminating Duplicates**

If you execute:

* Select type from titles;

You’ll get something similar to:

|  |
| --- |
| Buisiness |
| Buisiness |
| Cooking |
| Cooking |
| Cooking |
| Cooking |

Notice how there are multiple repeats of the same thing.

By using the distinct keyword, we can remove all duplicate values and only return unique or distinct values

* Select **distinct** type from titles;

This will instead return something like:

|  |
| --- |
| Business |
| Cooking |
| Fiction |
| Psychology |

*(Note no repeats/duplicates)*

## Where Clause – Selecting rows

The where clause can be used to select rows from a particular table.

The following techniques can be used to select particular rows from a table.

**Comparisons**

Logical expressions that test the range of values contained in the fields for each record. These may contain simple comparisons with constant values e.g.:

* Select au\_fname, au\_lname
* From authors
* Where contract = 1;

Or may compare the value of a computed expression with another computed expression

* Where advance \* 2 > ytd\_sales \* price;

Comparison operators are:

|  |  |
| --- | --- |
| Symbol | Meaning |
| = | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Not Equal to |

You can combine comparison operators using either **And** or **OR**.

E.g:

* Select title, price, adcvance
* From titles
* Where price > 10.00 and advance > 4000;

**Ranges (Between or** **Not Between)**

Ranges allow certain comparisons to be specified more neatly e.g.

* **Where** ytd\_sales >= 4095 **and** ytd\_sales <= 12000

Can be re-written as

* **Where** ytd\_sales **Between** 4095 **and** 12000

**Comapring strings with Between**

* Select pub\_name
* From publishers
* Where pub\_name **Between** ‘A’ **and** ‘F’;

***Note that no results with the letter ‘F’ e.g. “Five lakes publishing” are included as they appear after the letter ‘F’, think of a dictionary, all the words starting with F appear after the first appearance of the letter ‘F’.***

**Lists (IN, NOT IN)**

Lists with IN and Not In can be used to test set membership. In the simplest form you can specify a list of constant values enclosed in parenthesis:

* Where state IN (‘CA’, ‘IN’, ‘MD’)

You can also use a select statement whose results are used to specify the set:

* Select au\_lname, au\_fname from authors
* Where au\_id in (
  + Select au\_id from credits
  + Where royaltyper < 50
* );

*(nested select statements)*

**Pattern matches (Like, Not Like)**

Pattern matching with Like or Not Like allows wildcards to be used to match strings. The following wildcard characters are supported

|  |  |
| --- | --- |
| Wildcard | Meaning |
| % | Any string of zero or more characters |
| - | Any single character |

e.g. to test all phone numbers beginning with ‘028’, use:

* Where phone like ‘028%’;

**Unknown values (Is Null, Is Not Null)**

The statements Is Null and Is Not Null can be used to include or exclude rows where a particular column has an undefined or null value e.g.

* Where advance Is Null

Be aware of logical operator precedence (use brackets when unsure) Grouping Data & Data Aggregation

* Group by organizes data into groups
* **Having** sets condtions on which groups to include in the results (only use with ‘**Group by’**)
* Aggregate functions return summary values, either for the whole table or for groups within the table, they are therefore often used with **Group By**

|  |  |
| --- | --- |
| Aggerate function | Result |
| Sum (All | Distinct expression) | Totals all the values of numeric field |
| AVG(All | Distinct expression) | Averages all values of numeric field |
| Count(All | Distinct expression) | Numbers of values in the expression (non null) |
| Count(\*) | Number of selected rows |
| Max(Expression) | Highest value in the expression |
| Min(Expression) | Lowest value in the expression |

Example of aggerate function:

* Select pub\_id, sum(ytd\_sales\_ as Total
* From titles
* Group by pub\_id;

Or

* Select pub\_id, sum(ytd\_sales) as total
* From titles
* Group by pub\_id
* Having (count(\*) > 5);

**Order by- Sorting Query Results**

Sorting can be ascending or descending. In the simplest case you can write:

* Select pub\_id, type, title\_id
* From titles
* Order by pub\_id;

Or for descending order

* Select pub\_id, type, title\_id
* From titles
* Order by pub\_id desc;

Or specify the columns to order by bsed on the columns in the select

* Select pub\_id, type, title\_id
* From titles
* Order by 1 desc, 2 ,3;

**Joining tables**

Using join allows you to retrieve and combine data from two or more tables based on a relationsjop. You can and will typically join tables based on the relationships you define during database design, but you are not limited to using just these relationships.

Joins can be specified using either:

* The FROM clause with the JOIN key word
* Using a Where clause with a logical operator to specify how the tables should be joined.

An example of the first technique is:

* Select concat(au\_fname, ‘ ‘,au\_lname),
* Credits.au\_id, title\_id
* From
* Authors join credits
* On (authors.au\_id = credits.au\_id);

Note the following:

* The output column au\_id is qualified with the table name ‘Credits’ since the field name appears in two tables (authors & credits) specified in the from clause
* The tables authors and credits are joined using the expression authors.au\_id = credits.au\_id. All records in authors will be joined with records from credits where the au\_id fields are equal.
* Unambiguous fieldnames in the select clause do not need to be qualified

In the special case where the join column has the same name and data type in both tables, MySQL provides the using keyword to simplify the syntax;

* Select concat(au\_fname\_, ‘ ‘, au\_lname,
* Credits.au\_id, title\_id
* From
* Authors join credits
* Using(au\_id)

This highlights an important comparsion between using

* Authors join credits
* On (author.au\_id = credits.au\_id)

And

* Author join credits
* Using (au\_id)

In the case where a very complicated and populated database is being used, the former is recommended to be implemented as it clearly shows what columns are being selected from what tables.

In the case of the using statement, this is unclear unless the table structure of the database is very simple.

In order to increase readability, use On (tablename.x = table2name.y)

## Using Join

Types of join:

|  |  |
| --- | --- |
| Name of join | Description |
| Inner join | Returns only matches between the two tables |
| Left outer join | Returns matches between the two tables and also unmatched values from the ‘left’ table |
| Right outer join | Returns matches between the two tables and also unmatched values from the ‘right’ table |
| Full outer join | Returns matches between the two tables and also unmatched values from the left table then the right table |

Executing an inner join, because only those records where the columns being joined are equal are returned, this is known as an “inner join”.

The exact same result set can be produced by using a **where** clause to join the two tables:

* Select concat(au\_fname, ‘ ‘, au\_lname),
* Credits.au\_id, title\_id
* From authors, credits
* Where authors.au\_id = credits.au\_id;

*Same logical result as inner join through where statement*

**Joining multiple tables**

* Select concat(au\_fname, ‘ ‘, au\_lname), title
* From
* Authors join credits on authors.au\_id = credits.au\_id
* Join titles on credits.title\_id = titles.title\_id

*This links authors to titles through the credits table, effectively linking 3 tables*

**Outer joins**

So far inner joins have produced record sets containing records from multiple tables where column values are equal. Outer joins can be used to return records from either the left or right tables that do not have corresponding values in the other table being joined.

This allows you to answer questions like “list authors who have not written books”

* Select authors.au\_id, title\_id
* From
* Authors left join credits on
* Authors.au\_id = credits.au\_id;

*This executes a left outer join*

Note how this method returns more rows than the left inner join, this is because the additional records represent the authors who have not written books, i.e. those who do not have a match in the right table. For these records, the title\_id column is null.

**Semi-joins**

Occasionally you might want a list of records that are not matched within an inner join but appear only in an outer join. More specifically you might just want only those rows where the corresponding fields from one side of the join contain Null values, this is called a semi join.

* Select authors.au\_id , title\_id
* From authors left join credits on
* Authors.au\_id = credits.au\_id
* Where title\_id = Null;

You may expect this statement to return the authors who do not have any titles in the title table however no records are returned. This is because the columns from title\_id with null values do not actually exists, and so cannot be selected.

You can solve this problem by performing the final selection of rows with null values in the client application or rewrite the select statement

* Select au\_id from authors
* Where au\_id not in
* (select au\_id from credits);

Or alternatively

* Select au\_id from authors
* Where au\_id not in
* Authors left join credits on
* Authors.au\_id = credits.au\_id;

//TODO Test this

**Table alias**

You can see in all previous examples you need to qualify the column names with the correct table names when the same column name appears in more than one table, this can result in very long SQL statements.

You can alias the table names to shorter names (“A”,”B”, etc)

*Although this has a negative effect on the readability of the statement.*

* Select A.au\_id
* From
* Authors as A Join credits as B On
* A.au\_id = B.au\_id;

*A = authors & B = Credits*

**Unions**

There are times when you need to execute several SQL SELECT statements and return the records from these select statements as a single record set. This can be done using a **Union** statement with the following caveats.

* All the SQL statements must return the same number of columns
* All the columns must have the same data type.

Here is an example of a union that returns a list of authors and titles in the same record set

* Select ‘Author’ as thing, au\_id as ID,
* Concant(au\_lname, au\_fname) as Name
* From authors
* Union
* Select ‘title’, title\_id, title from titles;

## Views

Many database platforms support the idea of views, named queries that behave much like tables.

**Creating a view**

* Use world;
* Create view europeancities as
* Select a.name as country, b.name as city, a.region
* From country a join city by on a.code b on a.code = b.countrycode
* Where continent = ‘Europe’;

This creates a view called EuropeanCities with the structure defined by the query it contains. If you query the new view, you get a result set identical to that you’d get if you ran the query.

**Modifying Data**

The Insert, Update and Delete statements allow the data in tables to be manipulated. The operations performed by these statements are logged in the transaction log.

**Insert**

The insert statement allows records to be added to a table or view

* Insert Into tablename (ColumnList)
* Values (const1, const2)

*Note that ‘Into’ and ‘Column list’ are optional*

For example:

* Insert into publishers
* Values (‘9990’m’Jardin Inc’, ‘Camden’, ‘NJ’, ‘USA’);

In this case, values for all the fields that are non-null must be specified

**Working with Insert**

Execute this insert statement on the pubs database. In this next example values for only two fields are specified – it is assumed that the other fields in the table have default values or allow nulls

* Insert into publishers (pub\_id, pub\_name)
* Values (‘9991’, ‘The Health Center’);

The Insert statement also supports a nested select which allows records to be added by selecting them from existing tables i.e.

* Create table sfauthors LIKE authors
* Insert into sfauthors
* Select \*
* From authors
* Where city = ‘San Francisisco’;

If you want to create a table containing a copy of data from another table, you could also achieve the same result with a single statement.

* Create table sfauthors
* Select \* from authors
* Where city = ‘san francisco’;

**Insert with nested select**

Create a new table called “new authors” using this SQL statement

* Create table newAuthors(
* Name varchar(61) not mnull
* );

Execute this insert into statement to take records from the Authors table for authors in the state “CA”, concatenating their first and last names

* Insert intio newAuthors
* Select concat(au\_fname, ‘ ‘,au\_lname)
* From authors
* Where state = ‘CA’;

**Default values**

You can specify that the statement used default values for the columns:

* Insert into allemployee
* Values
* (
* ‘dfghjk’, ‘Katrina’, ‘L’, ‘thompson’,
* **Default**, **default**, **default**, ‘01/14/95’
* );

**Update**

This statement ius used to change data in existing rows in a table or view, either by adding new data or by modifying existing data

*The simple form of this statement is:*

* Update table-name
* Set column-name = expression
* Where search-conditions

For example:

* Update authors
* Set au\_lname = ‘Health’
* Au\_fname = ‘Goodbody’
* Where au\_lname = ‘Blotchet-Halls’;

Computed values can be updated, e.g.

* Update titles set price = price \* 2;

***Since no “where” clause is specified, all records in this table will be updated***

**Delete**

The delete statement is used to remove data from a table or view. Yhe syntax of this command is:

* Delete from tablename;
* Where clause;

**Note that if no “where” clause is specified, all records in this table will be deleted.**

## Summary

This chapter contained the most frequently executed statements in SQL, namely the *data manipulation language* statements.

These are the statements that applications and humans use to access the data contained within the tables.

They comprise a powerful set of syntaxes to operate at the statement level and also with individual values, whether contained within the table’s columns, supplied literally in the command or evaluated dynamically.

* **Select**: Not strictly DML by some specifications. This is by far the most commonly executed SQL statement, and it gives access to the data
* **Insert:** Adds new rows to the table
* **Update:** Modifies specified columns in existing rows
* **Delete:** Removes rows based on whether they match the condition in the where clause

Many clauses and syntaxes are common to multiple statements, so a good grasp of the select statement and its clauses such as how to evaluate expressions based on values and how to choose specific rows by using the where clause will help you write statements that access data in all sorts of complex and interesting ways.

# Programming in MySQL

## Overview

The database is where you store your data and the application is where you write your code. This is a general truth but there are times when you wan the database to do a little more than hold the data and perform some complex logic or flow control.

This is particularly the case when you have many different client applications accessing the same data, and you want to be sure that each one uses the same business rules when accessing the application. You might

* Design your permission structure so that no application user can modify table contents directly but must do so through stored procedures that you define in advance thus creating a simple API in the data layer.
* Create constraints implemented in triggers that apply hiusiness rules to the data based on complex conditions
* Execute regular operations that move or transform large amounts of data and place that logic in the database so that it doesn’t require network or socket traffic and row-by-row operation in a client application.
* Design operations that summarize or aggerate data so that the raw data never gets transmitted across a network for security reasons.

## Stored Routines

Stored routines are named blocks of code that you create once then execute repeatedly as you would built-in statements or functions. This gives you the benefit of a named API and means you limit code duplication.

Each stored routine has a name and a body.

Two types of stored routine:

* Stored procedures are statements, executed as an appropriate **CALL**  command (e.g. CALL Set\_Test\_Data(); )
* Stored functions are treated as expressions and are used within statements (e.g. Count(\*);)

Typically you create a stored procedure when you want to wrap an operation within an API. For example you could create a procedure to insert a record that first checks that the values are correct, then inserts the record and finally logs that transactio.

Similarly you could create a function that creates a value based on other values or values from the database. E.g. an invoice code might consist of a date portion concatenated with a customer code and a sequence i.e. “1307DAN003’”, this value would be calculated from a stored function without requiring a round trip to the application code.

In order to create a stored routine, **you must have the CREATE ROUTINE privilege.**

## Stored Procedures

The **Create Procedure** statement creates a procedure in the default database.

It requires the name of the new procedure and a parenthesized list of its parameters which can be empty. Then you provide the statement or compound statement that forms the body of the procedure.

For example, in the world database

* **Create procedure** CityCount()
* **Select Count(\*) from** city;

*Note this procedure has no parameters, empty parentheses*

**To remove a procedure,** use the **DROP** statement.

*Note: throws an error if procedure does not exist*

* **Create Procedure** ProcToDrop() **Select 1;**
* **DROP Procedure** ProcToDrop;
* **DROP Procedure** ProcToDrop;
* **DROP Procedure** ProcToDrop **If not exists;**

*First drop succeeds*

*Second drop fails (procedure doesn’t exist by that stage)*

*Third succeeds (If not exists prevents drop from executing erroneously)*

Use **Call** to execute a procedure:

* **Call** CityCount;

For more complex procedures, the parentheses in the declaration can contain parameter declarations consisting of the name and type of the parameter.

* **Create procedure** CitieiesInCountry(code **char(3**))
* **Select Count(\*)** **From** City;
* **Where** CountryCode = code;

To execute a procedure that takes parameters you must provide an argument of the appropriate type when calling the procedure

* Call CitiesInCountry(‘GBR’);

If you don’t provide a value for a parameter required by a procedure, you get an error message.

For even more complex procedures, you can specify that it requires more than one parameter. Each parameter has a name and a type, separated by commas:

* Create procedure CityPopBetween(low in, high int)
* Select Name, Population FROM city
* Where Population Between low and high;

To call such a procedure, a value must be provided for each of the defined parameters.

## The Definer Attribute

You can create procedures that execute statements the user would not normally be allowed to execute. For example, you might configure privileges so that your application user cannot insert data, but they could add rows if they used a stored procedure.

The procedure then can use logical checks on the data to ensure all values are appropriate and log the fact that the insert took place.

To configure this, you can create a procedure with the “DEFINER” clause referencing the identity of the account whose privileges the procedure adopts when executed by any user.

* Create
* Definer = bob@localhost
* Procedure CitiesInCountry3(code char(3))
* ……. (rest of procedure declaration)

Every user who executes this procedure will do so with the permissions of bob.

**Important note:** If no definer is declared, it defaults to **the account used to create the procedure**, I.E. if you are signed in as admin and create a procedure, anyone who executes the procedure will do so as admin.

To avoid this either include a definer for an account that is okay to execute said command or use the **security invoker** clause

e.g.

* Create procedure CitiesInCountry(code char(3))
* SQL SECURITY INVOKER
* ……. (rest of procedure)

The security invoker allows the procedure to be executed as the user who is executing, so even though Admin@localhost made the procedure and Bob@localhost is executing it, it will be executed under Bob’s privileges.

Think of security invoker as the definer attribute, except each time the procedure is ran, the definer is set to the user who is running the command.

## Stored Functions

Functions are used within statements as expressions. As such they can be called from within any statement, anywhere you would provide a value of the function’s return type.

To declare a function, use the **create function** syntax:

* **Create function** square(I int)
* **Returns int**
* **return i\*I;**

The returns clause specifies the type of the return value. It is followed by the body of the function, a statement or compound statement which must contain a return statement

Invoke a function by using it within a statement

e.g.

* SELECT Square(5);

## Programming Constructs

MySQL supports basic conditional processing, looping, flow control and other programming constructs you might expect from a fully-fledged programming language.

## Compound statements

Stored routines within MySQL behave like single statements to the calling code but frequently contain many individual statements. Even within a stored routine you could have a conditional loop that iterates over multiple statements. A compound statement is a set of statements taken together as a single entity for the purpose of programming.

## Multiple statement problem

Compound statements consist of one or more statements enclosed within the BEGIN and END markers. Compound statements are themselves considered statements that must be terminated

*This code throws an error*

* Create procedure BroeknProc(code char(3))
* Begin
* Select count(\*) from city
* Where countrycode = code;
* Select now();
* End;

*This throws a syntax error if you try to execute it*

The reason is due to thee facrt that the client submits the procedure after the line marked (where countrycode = code) because it sees the semi colon and thinks it is the end of the statement, however it should submit the statement after the “end”.

To solve this problem we use the **Delimiter**

We are able to change the statement terminator by using the Delimiter keyword, think of it as a *temporarily changing the semi colon to something else, then changing it back just after the statement is submitted to the server*

This allows the client to parse the entire procedure, semi-colons and all, before passing it to the server.

e.g.

* **Delimitter //**
* **Create procedure** CitiesInCountry(code **CHAR(3)**)
* **Begin**
* **Select Count(\*)**  from city
* **Where CountryCode = code;**
* **Select now()**
* **End //**
* **Delimitter ;**

*Note, you don’t have to use “//”, you can use any valid sequence that doesn’t appear within the procedure*

## Variables

MySQL has a number of different types of variables available to the DBA and developer. Some of these are system or status variables, others are available when writing code

**User variables**

User variables are similar to variables in other languages in that they are containers for values and have a name and a data type.

In MySQL user variables are named according to the standards for other identifiers and are prefixed with an @ sign. Such variables are only available within the session they are created.

The Type can change throughout the line as well as the value.

Assign to variables using the := syntax

* SET @var1 := 2;
* SELECT @var2 := 4;

You can subsequently read the value of a variable in any statement that accepts and expression of that variable’s type:

* Select @var1, @var2, @var1 + @var2;
* Select Name, Population/@var1 from city
* Where countrycode = ‘IRL’
* Insert into table1(a,b) values(@var1, @var2);

You can use the INTO keyword to assign values from a select statement into variables e.g.

* Select Name into @zwe from country where Code=’ZWE’;
* **Select MAX(**Population) into @Maxpop from city;
* **Select @maxpop;**
* **Select name into @biggestcity from city**
* **Where population = @maxpop;**
* **Select @biggestcity, @maxpop;**

This shows that variables values are persistent across the same session but not across different sessions.

**Local variables**

You can also declare variables that are local to a compound statement by using the declare statement. Variables so declared do not have an “@” prefix.

e.g.

* Delcare five INT;
* Set five := 5;

**Difference between user variables and local variables**

Local variable is set to null after a procedure has been called, whereas user variables persist during the session and are set to null when their session ends.

## Conditional processing

The primary decision-making constructs in MySQL are IF and CASE, these key-words work as both statements and expressions.

**If statement and Expression forms**

The if statement as in other languages executes a statement if the provided condition is true. Use THEN and ELSE clauses to specify the statement to execute if the condition is true/false respectively.

You can also use the else-if clause in conjunction with its own **then** to string together subsequent conditions. Finally use **end** **if** to terminate the conditional block.

e.g.

* Delimiter //
* Create procedure less\_than\_five(input INT)
* Begin
* Declare five INT;
* Set five := 5;
* It input < five then select ‘Input is less than five’;
* Else select ‘Input is not less than five’
* End if
* End //
* Delimiter ;

The IF function is a form of ternary conditional operation. You provide three arguments, the function evaluates the first argument, returning the second if the first evaluates as true and the third is false. As a function you can use it within any statement that accepts a value of its return type. That is of the same type as its second and third arguments.

**Case**

The CASE statement contains a number of other statements, one of which is executed based on the condition specified. The body of the statement comes between the CASE and END CASE keywords and each conditional statement is prefixed with a WHEN…THEN combination, specifying the condition or it’s match.

Two main forms of the CASE statement, in each case, a comparison is made with the difference being in how the value is compared.

* Simple case, where a value is checked for equality with another value
* Searched case, where a Boolean expression is executed

**Simple Case**

* **Case x**
* **When y then statement\_1;**
* **When z then statement\_2;**
* **Else statement\_3;**
* **End case**

**Searched case**

* **Case**
* **When comparison\_1 then statement\_1;**
* **When comparison\_2 then statement\_2;**
* **Else statement\_3;**

Note there shouldn’t be any values or statements between “Case” and the first “WHEN”

The CASE Expression can be used inside a statement and evaluates as a single value.

## Looping (Brother give LööP)

MySQL also supports familiar looping constructs:

* *Loop*…*End* *loop* simply repeats until the loop explicitly exits using the *Leave* or *Return* statements
* *While* **condition** *Do* … *End* *While* allows conditional test-first looping
* *Repeat* … *until* conditions *End* *Repeat* allows conditional test-last looping

Inside a loop you can also use the following loop control constructs

* *Iterate* restarts the current iteration of the loop
* *Leave* exits the loop or compound statement
* *Return* exits the stored function.

e.g. the following examples creates a procedure that uses a basic *Loop* … *End* *Loop,* loop to count from one to ten.

* **Create** **procedure** counter()
* **Begin**
* **SET** @a := 0;
* main: **LOOP**
* **select** @a := @a + 1;
* **if** @a >= 10 **then**
  + **leave** main;
* **End** **IF**;
* **Iterate** main;
* **End** Loop main;
* **End**

# Triggers

A trigger is a named block of code that executes automatically when the triggering event fires.

The triggering event is a data modification statement – **Insert**, **Update** or **Delete** on a given table.

## Creating Triggers

When you create a trigger, you specify

* The trigger’s name
* Whether it executes **before** or **after** the operation that causes it to fire
* The data modification statement – **Insert**, **update** or **delete** that causes it to fire
* The table it belongs to
* The trigger body, i.e. the code it executes

We will create a basic logging trigger

*Create a table in the world database to contain the log by issuing the following commands*

* Use world;
* Create table CityLog(
* Operation varchar(50),
* What int,
* Who varchar(50),
* Mod\_time DateTime);

*Create a trigger to record all new records going into the city table*

* Delimiter //
* Create trigger log\_city\_additions
* After insert on City
* For each row begin
* Insert into CityLog(Operation,What,Who,Mod\_time)
  + Values(‘New City added’, NEW.ID, USER(), NOW());
* End //
* Delimiter ;

*Insert some new cities using statements like:*

* Insert into city VALUES (4080, ‘Waterford’, ‘IRL’,’Munster’,80000);

*Select from the new log table to verify that the trigger is working*

* Select \* from CityLog;

When creating triggers keep the following points in mind:

* You can only have on trigger per table per operation per firing time (Before or After)
* Triggers that fire on **INSERT** also fire for **LOAD** **DATA** and **REPLACE** operations
* Triggers that fire on **DELETE** also fire for **REPLACE** operations.
* The **OLD** and **NEW** table aliases refer to outgoing and incoming versions of rows respectively.
* In general, use BEFORE triggers to calculate row values for insertion or to evaluate constraints and thus prevent the row operation taking place and use after triggers for logging or updating summary tables

**SHOW** **TRIGGERS** FROM world; (*displays all triggers in a db).*

## Using Triggers

Triggers are most useful when you want the database to perform some action whenever data changes in some way, regardless of the client or user that imitated the action. In particular triggers are used for the following purposes.

* Calculating values during insertion e.g. if you have a price field in an orderItems table, and a product\_ID column referencing a product in another table that has a list price field, you can calculate a value for a discount field by comparing the two in a trigger.
* Calculating values for summary e.g. if you have many thousands or millions of operations per day but you have frequent reports or user interface fields that read sum totals, you can have a trigger that creates summary or historical data for things like page counts or total order count and values for the day or week.
* Maintaining domain integrity with complex rules for example registration numbers, national insurance numbers (IMHO creating a procedure that must be called to insert a value that checks the input value before trying to insert is more secure but triggers are a brilliant catch all that prevents even people circumventing such a defined procedure) (best to have both)
* Logging data modifications
* Implementing ACL based access security at a row level (what does ACL mean?)
* Maintaining historical data in a data warehouse with surrogate keys and entity versioning.

This is by no means an exhaustive list but should give you a flavor of the flexibility triggers afford in complex data environments.

## Summary

Stored programs allow you to create blocks of code that operate within the database.

They are particularly useful when you want operations to take place entirely in the data tier for security or efficiency reasons or when you have operations that should take place the same way regardless of which client application executes them.

Stored programs are named blocks of code that perform repeatable functions.

Stord routines-procedures and functions are named programs that operationally take parameters.

Stored procedures execute as statements whereas stored functions execute as expressions and must therefore return a value.

Triggers are stored programs that execute automatically when a certain operation occurs on a specified table.

When writing stored programs you can make use of an extended syntax that allows for common programming constructs such as variables, conditional processing and looping.

# Optimizing performance

## Overview

The performance of a database relies on many different interrelated things. Some of the environmental factors that affect a server’s performance are:

* The operating system – other applications, general server performance, virus utilities, backup tools and other concurrent activity
* It’s virtualized state – other VMs on the host can slow down the hardware
* The speed of the network it runs within whether there are many clients issuing small requests or large single-purpose clients transferring lots of data
* It’s hardware: RAM, CPU, hard disk subsystem (throughput and seek time) some applications like reading small amounts of data in a random-access fashion whereas others like reading large chunks of data in one go.

## Performance implications of database design

The design of a database also plays a key role in how it performs under load. If you have a database that is not normalized then it likely suffers from redundancy or high coupling between its parts.

This means the application has to spend effort in keeping everything connected

Non optimal data types can increase the I/O load on a server, if you use BIGINT when SMALLINT would do then that column alone uses 4x the effort & data.

## Performance implications of queries

You can also improve the performance of a database by ensuring that queries are correctly structure, that indexes are created to suit those queries and that the server is tuned for its workload.

Applications that make use of transactions can suffer performance problems with locking between concurrent transactions and the potential for deadlocks is one that should be carefully considered.

## Identifying queries to tune

MySQL has a slow query log that records all statements lasting over a specified duration. This makes it very useful when tuning a database’s performance because it helps you identify those queries that take the longest amount of time to run. It is configured with server options.

* Slow\_query\_log enables the slow query log
* Long\_query\_time The minimum time (in seconds, with microsecond precision) a statement must take to be logged
* Log\_slow\_admin\_statements enables the logging of admin statements they’re not logged by default
* Log\_queries\_not\_using indexes enables the logging of queries that do not use indexes.

You should also bear in mind that many applocations issue the same (or similar) statements many thousands or millions of times per day, and that tuning a statement that takes 200ms to run but runs 1,000,000 times a day will have a greater affect than tuning a statement that takes 5 seconds to run but is only run a couple of times per day.

## Indexes

Indexes help the optimizer find rows more easily. Each table that has a primary key is organized in that order. This means that rows are stored in the table on the file system in the same order as their primary key value.

You can also create secondary indexes on other fields to help queries that rely on those fields. Each seconday index contains the fields specified, plus a copy of the primary key so that the optimizer can find the row it refers to. This is similar in concept to the indexes you’d find at the back of a technical book.

**The explain command**

The exmplain command is useful to see how MySQL’s optimizer executes a statement. In particular Explain shows which keys (indexes) are used by the query

*Example output*

* ***Explain******Select name FROM city WHERE ID = 5\G***
* *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
* *Id : 1*
* *Select\_type: SIMPLE*
* *Table: city*
* *Type: const*
* *Possible\_keys : Primary*
* *Key : Primary*
* *Key\_len : 4*
* *Ref : const*
* *Rows: 1*
* *Extra :*

This query uses the primary index, because the where clause used the ID field to uniquely identify the riow and that field is the primary key field, The query compares that field to a constant, the value “5”.

**The following output shows a query that does not use an index:**

* ***Explain select name from city where population =215363\G***
* ***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****
* *id : 1*
* *select\_type : simple*
* *table : city*
* *type : all*
* *possible\_keys : null*
* *key : null*
* *key\_len : null*
* *ref: null*
* *rows : 4051*
* *extra : using where*

This query cannot use any indexes; the possible\_keys field(which can list several keys if there are any that are suited) does not list any indexes and the key field shows the optimizer has not chosen any such key. The outcome is that the optimizer needs to read 4051 rows in order to arrive at the row requested.

## Creating indexes

To create an index, use the CREATE INDEX statement, specifying the name of the index and the column(s) it contains, for example:

* Create Index city\_pop on city(population)

This creates a secondary index on the city table, ordered on the population field. Queries that use this field can use the index as a possible key and if the optimizer decides that the key is more optimal than any other (including the primary key) it uses it.

After creating this index, the output of “Explain” changes accordingly

* ***Explain select name from city where population =215363\G***
* ***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****
* *id : 1*
* *select\_type : simple*
* *table : city*
* *type : ref*
* *possible\_keys : city\_pop*
* *key : city\_pop*
* *key\_len : 4*
* *ref: const*
* *rows : 1*
* *extra :*

After creating the city\_pop index, the optimizer decides that it is a good candidate for the query.

If your query is made up of multiple statements then each one has an entry in the output of the “Explain” command.

## Tuning

Tuning is the act of simply trying something out, changing some configuration and trying it again to see if there is an improvement, it works for guitar strings and it works for databases.

**Using MySQLSlap**

MySQL comes with a utility called MySQLSlap that allows the same query to be repeatedly executed and measures the time it takes to run.

Consider the following command

* **mysqlslap -uroot –create-schema=world \**
* **--query=”select name from city where population=215363;” \**
* **--concurrency=20 –iterations = 20**

*Benchmark*

*Average num of second: 0.006 seconds*

*Minum num of second: 0.004 seconds*

*Max num of second: 0.010 second*

*Number of clients: 20*

*Average number of queries per client: 1*

The options shown are:

* -uroot, the root user, you may need to include a password, for example -ppassword
* --create-schema=world this is the database mysqlslap connects to to run it’s query
* --query, the query that is run for the bench mark
* --concurrency How many concurrent clients mysqlslap runs
* --iterations how many times each client runs the query

## Using SQL\_NO\_CACHE

For small tables the entire dataset can be stored in the buffer pool. An area of memory used to store recently-accessed data.

* **mysqlslap -uroot –create-schema=world \**
* **--query=”select SQL\_NO\_CACHE name from city where population=215363;” \**
* **--concurrency=20 –iterations = 20**

In general, an optimization that improves performance on benchmarks that use SQL\_NO\_CAHCE performs better overall, without SQL\_NO\_CACHE you might only be testing the performance of the buffer pool.

## Tuning indexes

To improve this query we could create an index on the population field of the city table as described earlier.

Now running the same benchmark (retaining the SQL\_NO\_CAHCE option to avoid the benefit of the buffer pool) gives better results