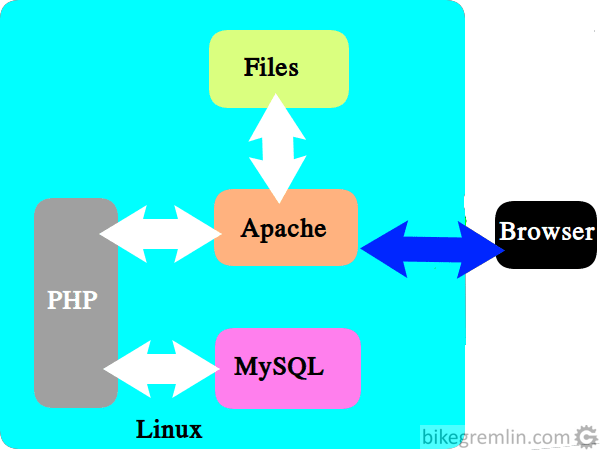
# What is LAMP stack?

LAMP is, to oversimplify it, an acronym for Linux Apache MySQL PHP. It is an environment for developing and deploying web applications and websites.

* + Linux is the operating system on which it all runs.
  + Apache is the web server (though Nginx can also be used instead).
  + MySQL (or MariaDB) is a relational database management system.
  + PHP (or Python, Perl) is the language for running scripts.



# Understanding Ports

* A network connection between two devices in a network requires a socket at each end and a connection (or a physical port).
  + Socket - a combination of an address and a port
    - Example: 0.0.1.10:22
    - Example: 10.0.1.11.53182
  + Connection -
    - Example: 10.0.1.10:22
    - Example: 10.0.1.11:53182
* Port - a logical space where traffic or communication for a particular process or service is intended to go.
  + Their range is from 0 to 65,535. This allows many services to listen and communicate on one network interface.

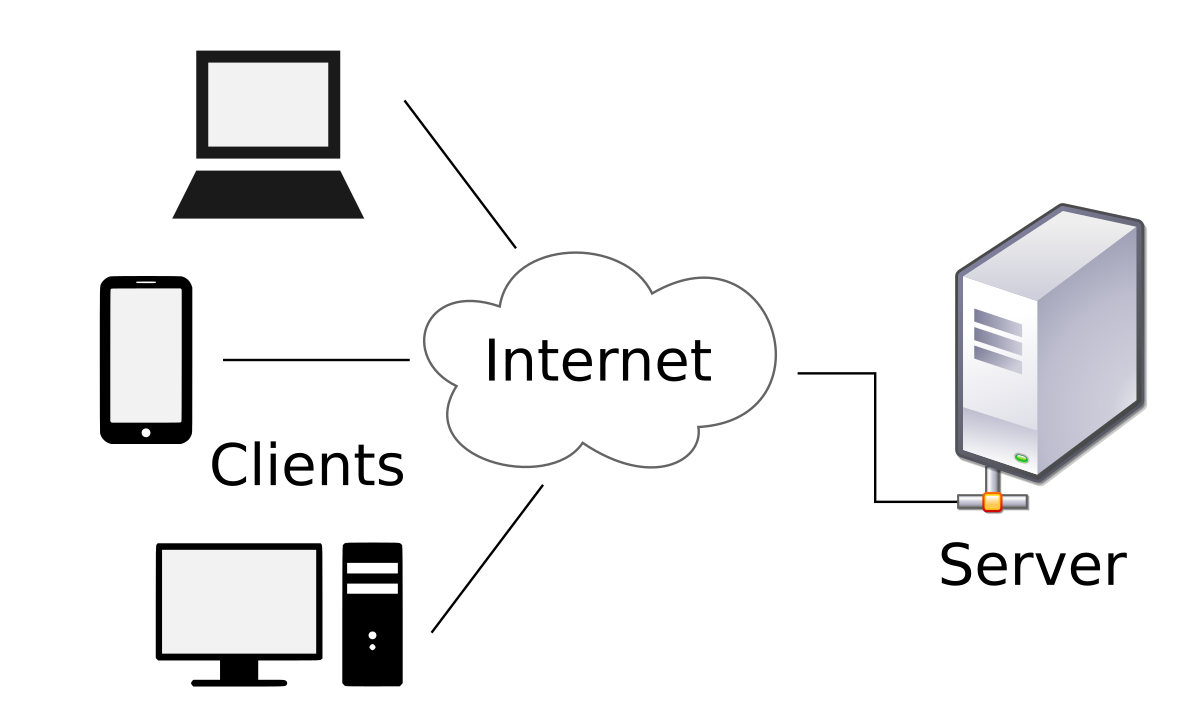
| Range | Name | Use |
| --- | --- | --- |
| 0-1023 | Well-Known Ports | Common services (22:ssh, 80:http) |
| 1024-49151 | Registered Ports | Unprivileged services (development servers) |
| 49151-65535 | Dynamic Ports | Outgoing connections to other systems |

* For each port a processor can listen using TCP and/or UDP protocol. A process can control more than one port. A port can only be used by one process at a time.
  + TCP - Transmission Control Protocol
  + UDP - USer Diagram Protocol

# An Introduction to MySQL

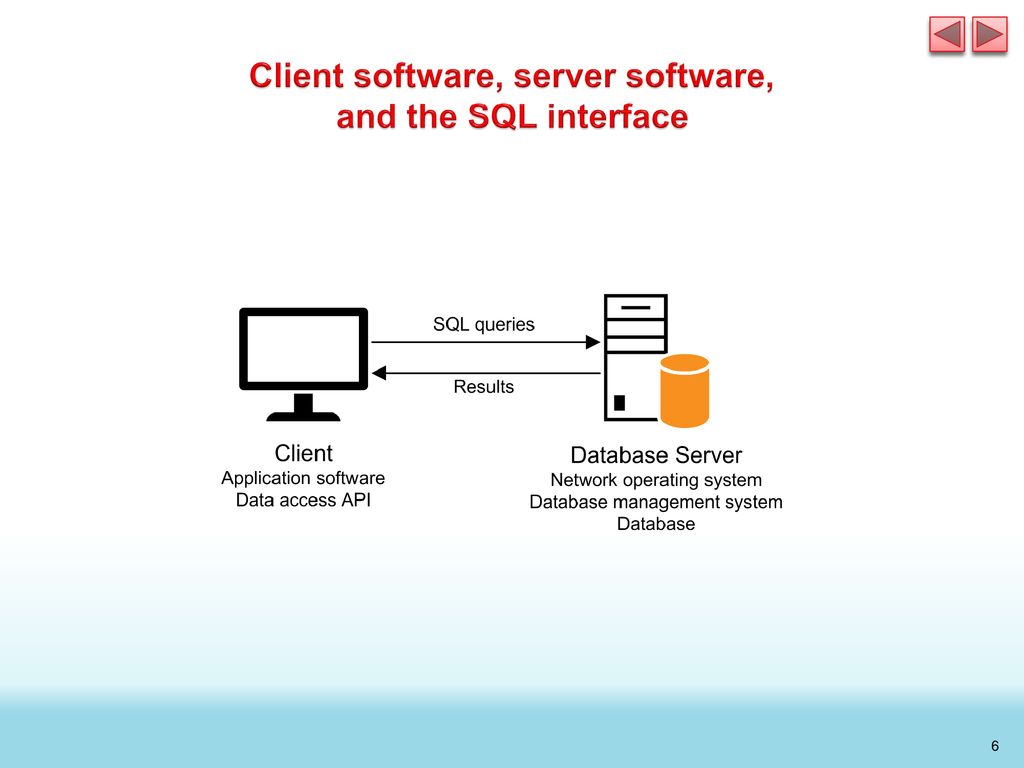
## Chapter 1: An Introduction to Relational Databases

### A simple client/server system



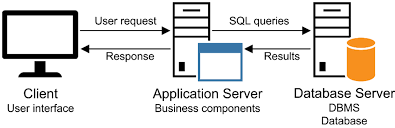
The three hardware components of a client/server system

* + Clients - PCs, Macs, or workstations of the system
  + Server - store files and databases of the systems for clients to use
  + Network - a way to electronically communicate between clients
* Terms to know about client/server systems
  + Local area network (LAN) - a computer network that interconnects computers within a limited area
  + Wide area network (WAN) - telecommunications network that extends over a large geographic area
  + Enterprise system - any kind of information system which improves the functions of enterprise business processes by integration

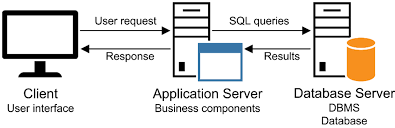


* Server software
  + Database management system (DBMS) - allows us to store and manage the databases of the client/server system
    - The DBMS does the back-end processing.
    - MySQL is an example
* Client software
  + Application software - does the work the user wants
  + Data access API (application programming interface) - provides an interface between the application program and the DBMS
    - The client software does the front-end processing
* The SQL interface
  + The application software communicates with the DBMS by sending SQL queries through the data access API.
  + When the DBMS receives a query, it provides a service like returning the requested data (the query results) to the client.
  + SQL stands for Structured Query Language, which is the standard language for working with a relational database.

A networked system with an application server



A simple web-based system

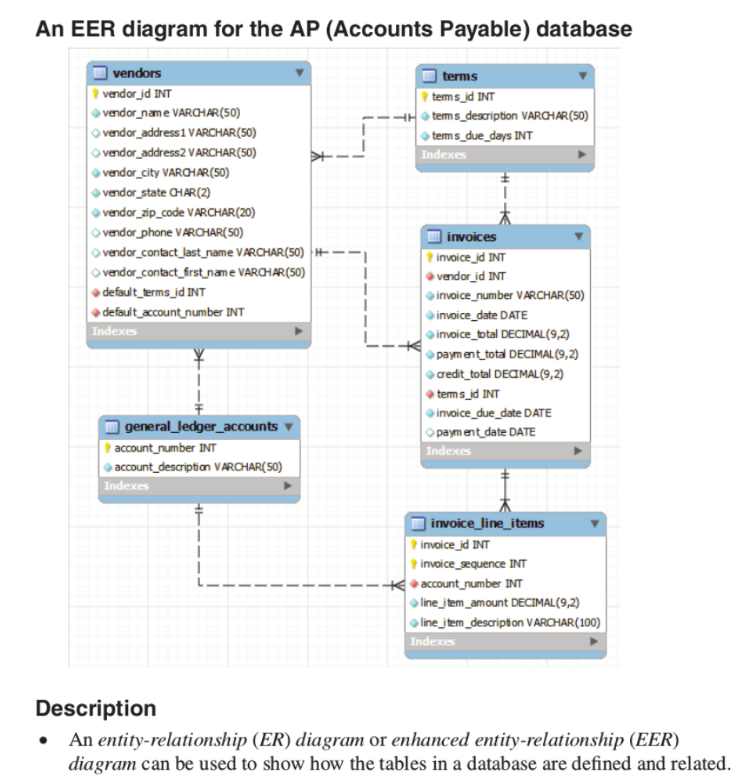


* Terms to know about database tables
  + Relational database - consist of tables
  + Primary key - uniquely identifies each row in the table
  + Composite primary key - a primary key that uses two or more columns
  + Non-primary key (unique key) - basically the same thing as a primary key
* Terms to know about table relationships
  + Foreign key - identifies a related row in the Venders table
  + Referential integrity - property of data stating that all its references are valid
  + One-to-many relationship - occurs when each record in Table A may have many linked records in Table B, but each record in Table B may have only one corresponding record in Table A
  + One-to-one relationship - when one row in a table may be linked with only one row in another table and vice versa
  + Many-to-many relationship - a relationship between tables in a database when a parent row in one table contains several child rows in the second table, and vice versa

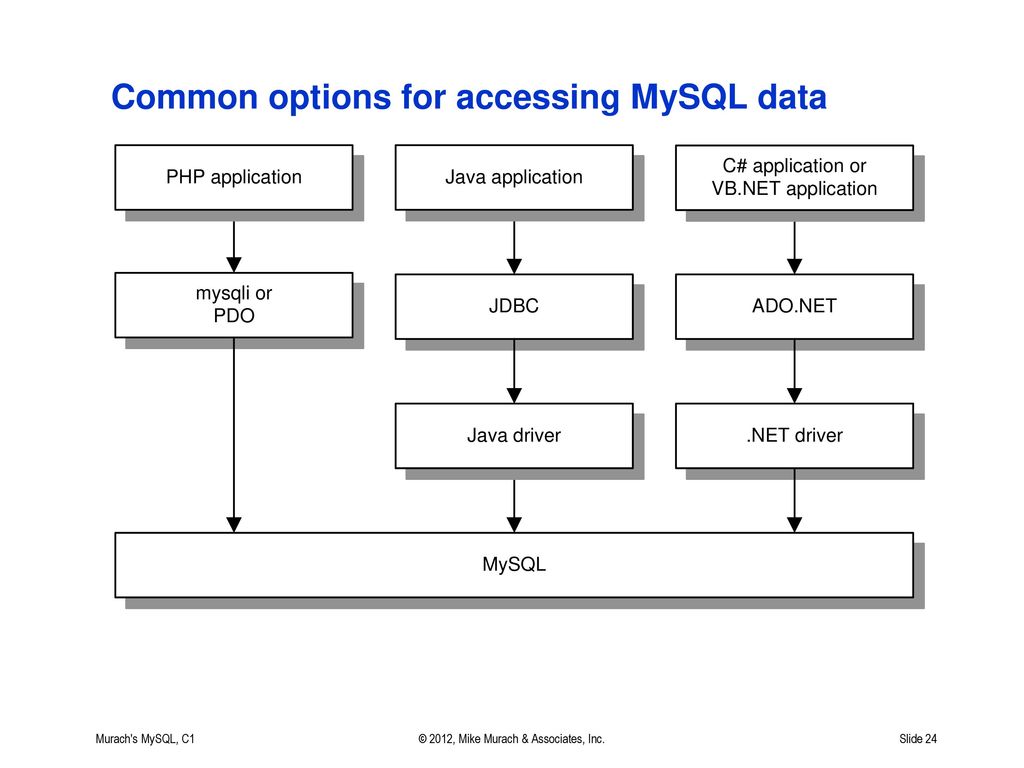
### Common MySQL data types

* CHAR, VARCHAR - string of letters, symbols, and numbers
* INT, DECIMAL - integer and decimal numbers that contain an exact value
* FLOAT - floating point numbers that contain approximate value
* DATE - Dates and time

An EER diagram for the AP database



* SQL DML statements
  + SELECT - Retrieves data from one or more tables
  + INSERT - adds new rows to a table
  + UPDATE - changes existing rows in a table
  + DELETE - deletes existing rows from a table
* SQL DDL statements
  + CREATE DATABASE - creates a new database on the server
  + CREATE TABLE - creates a new table in a database
  + CREATE INDEX - creates a new index for a table
  + ALTER TABLE - changes the definition of an existing table
  + ALTER INDEX - changes the structure of an existing index
  + DROP DATABASE - Deletes an existing database and all of its tables
  + DROP TABLE - Deletes an existing table
  + DROP INDEX - Deletes an existing index
* Terms to know about SQL
  + Query - a SELECT statement
  + Base table - you use the SELECT statement to retrieve selected columns and rows
  + Result table (result set) - the result of SELECT
  + Join - lets you combine data from 2 or more tables into a single result set
  + Inner join - returns rows from both tables only if their related columns match
  + Outer join - returns rows from one table in the join even if the other table doesn’t contain a matching row
* Comment
  + /\*block comment\*/
  + -- comment



## Chapter 2: How to use my SQL Workbench and Other Development Tools

### Terms

* MySQL Workbench - a free graphical tool that makes it easy to work with MySQL
* Database Server - it receives SQL statements that are passed to it, processes them and returns results.
  + Database Service - a cloud computing managed service offering that provides access to a database without requiring the setup of physical hardware, the installation of software, or the need to configure the database.
  + Database engine - the underlying software component that a database management system uses to create, read, update and delete data from a database.
* Database object - a data structure used to either store or reference data
* Schema - contains tables that store information required by the MySQL server as it runs
* Snippet - a template containing the basic structure of a Transact-SQL statement or block
* SQL Script - a file that contains one or more SQL statements

## Chapter 3: How to Retrieve Data from a Single Table

### Keywords

* Keyword - Capitalized words
  + SELECT, FROM, WHERE, etc…

### Arithmetic operators

| Operator | Name | Order of precedence |
| --- | --- | --- |
| \* | Multiplication | 1 |
| / | Division | 1 |
| DIV | Integer Division | 1 |
| % (MOD) | Modulo (remainder) | 1 |
| + | Add | 2 |
| - | Subtract | 2 |

* Order of precedence - left to right based on the order value
* Comparison operator

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

### Basic syntax of the SELECT statement

| SELECT  Select list  FROM  Table source  WHERE  Search condition  ORDER BY  Order by list  LIMIT  Limit the number of rows |
| --- |

### SELECT

* SELECT - describes the columns in the result set
  + Select list - the columns of the base table you want

| SELECT column\_name1, column\_name2, etc… |
| --- |

* + Expression - you can use arithmetic expressions to make a new column
  + Column Alias - use the word AS to specify the name for the column in the result set
    - payment\_date AS Date Payed
    - invoice\_total - payment\_total AS remanding\_amount

| SELECT select\_list, column\_name3 AS new\_name |
| --- |

OR

| SELECT select\_list, expression AS new\_name |
| --- |

* + CONCAT(column1, column2, etc…) AS new\_column - a function that concatenates columns together and the variables in those columns

| SELECT select\_list, CONCAT(column\_name4, column\_name5), etc… |
| --- |

* + LEFT - extracts the letters starting from the left of the string to the number of characters needed

| SELECT select\_list, LEFT(column\_name(has to be a string value), number\_of\_characters), etc… |
| --- |

* + DATE\_FORMAT - changes the format of the date
    - MM/DD/YY = ‘%m/%d/%y’
    - DD-Mon-YYY = ‘%e-%b-%Y’

| SELECT select\_list, DATE\_FORMAT(column\_date, ‘format’), etc… |
| --- |

* + ROUND - rounds the value of the function to the nearest value specified after the decimal (if the number is 2, rounds to the nearest 10^-2)

| SELECT select\_list, ROUND(column\_int3, number), etc… |
| --- |

* + DISTINCT - a keyword you put right after SELECT keyword that prevents duplicate rows from the result set

| SELECT DISTINCT select\_list |
| --- |

* + ALL - a keyword you put right after SELECT keyword that includes all matching rows in the result set regardless if they are duplicated

| SELECT ALL column\_name1, column\_name2, etc… |
| --- |

* + Wildcards:
    - \* - selects all columns

### FROM

* FROM - Names the base table from which the query retrieves the data
  + Table source - The table name

| FROM base\_table |
| --- |

### WHERE

* WHERE - specifies the conditions that must be met for a row to be included in the result set
  + Search condition - this is to filter the rows from the base table that when the search condition is true it is included in the result table
  + Logical operator - AND and OR operators than combine two or more search conditions
    - You can use a NOT operator to negate a search condition

| WHERE [NOT] search\_condition1 {AND|OR} search\_condition2 … |
| --- |

* + IN phrase - used to test whether an expression is equal to a value in a list of expressions

| WHERE test\_expression [NOT] IN ({subquery|expression\_1 [, expression\_2]...}) |
| --- |

* + BETWEEN phrase - to test whether the condition falls within a range of values

| WHERE test\_expression [NOT] BETWEEN start\_value AND end\_value |
| --- |

* + LIKE phrase - lets you search for simple string patterns
    - LIKE wildcards:
      * % - matches any string of zero or more characters
      * \_ - matches any single character

| WHERE match\_expression [NOT] LIKE pattern |
| --- |

* + REGEXP phrase - allows you to create complex string characters
    - REGEXP wildcards:
      * ^ - matches the pattern to the beginning of the value being tested
      * $ - matches the pattern to the end of the value being tested
      * . - matches any single character
      * [charlist] - matches any single character listed with the brackets
      * [char1-char2] - matches any single character within the given range
      * | - separates two string patterns and matches either one

| WHERE match\_expression [NOT] REGEXP pattern |
| --- |

* + IS NULL phrase - returns if something has a null value

| WHERE test\_expression IS [NOT] NULL |
| --- |

### ORDER BY

* ORDER BY - Specifies how to sort the rows in the result set. This keyword is allowed to use column alias’
  + Order by list - the order in which the rows are to be sorted
  + ASC - ascending order (default), put after an order list
  + DESC - descending order, put after an order list

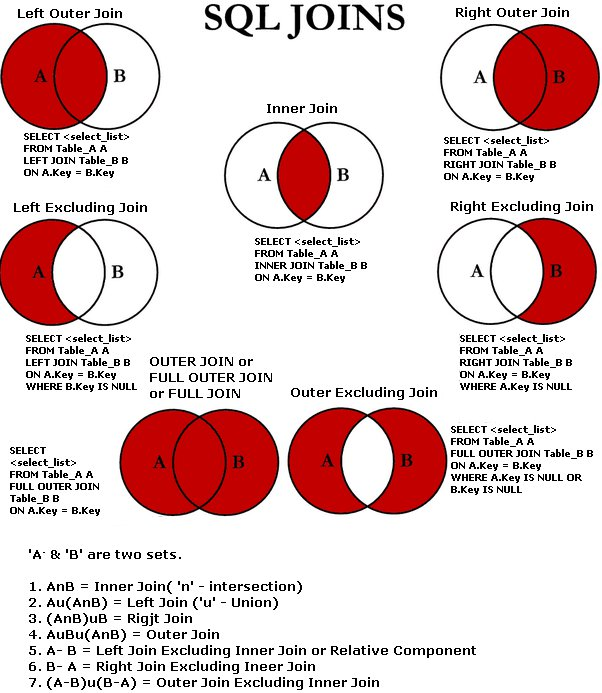
| ORDER BY column\_name [ASC|DESC] |
| --- |

* LIMIT - Specifies the number of rows to return
  + row\_count - the number of rows
  + Offset - starts with the offset index (all rows start with index 0)

| LIMIT [offset,] row\_count |
| --- |

## Chapter 4: How to retrieve data from two or more tables

### INNER JOIN

****

* Join - Lets you combine columns from two or more tables into a single result set
* Join condition - indicates how two tables should be compared
  + ON specifies the join condition
  + Inner join - only those rows that satisfy the join condition are included in the result set

| SELECT select\_list  FROM base\_table  [INNER] JOIN table\_1  ON join\_condition\_1  [[INNER] JOIN table\_2  ON join\_condition\_2] … |
| --- |

* Qualified column name - you enter the table name and a period in front of the column name
* Table alias - alternative table name that’s typically a letter or two

| SELECT select\_list  FROM base\_table a1  [INNER] JOIN table\_1 a2  ON join\_condition\_1  [[INNER JOIN table\_2 a3  ON join\_condition\_2] … |
| --- |

* Schema - a collection of databases

| FROM … database\_name.table\_name … |
| --- |

* AND OR conditions

| SELECT select\_list  FROM base\_table a1  [INNER] JOIN table\_1 a2  ON a1.column\_name1 operator a2.column\_name2  [AND|OR] a1.column\_name3 operator a2.column\_name4 |
| --- |

* Self-join - joins a table to itself

### OUTER JOIN

* Outer join - returns all of the rows from one of the tables involved in the join, regardless of whether the join condition is true
  + Left outer join - the first table
  + Right outer join - the second table

| SELECT select\_list  FROM base\_table  {LEFT|RIGHT} [OUTER] JOIN table\_1  ON join\_condition\_1  [{LEFT|RIGHT} [OUTER] JOIN table\_2  ON join\_condition\_2] … |
| --- |

### EQUIJION

* Equijion - when you use an equal operator to join two tables on a common column
  + USING - use this instead of ON when comparing columns (usually of the same name)

| SELECT select\_list  FROM base\_table  [{LEFT|RIGHT} [OUTER]] JOIN table\_1  USING (join\_column\_1[, join\_column\_2]...)]...  [[{LEFT|RIGHT} [OUTER]] JOIN table\_2  USING (join\_column\_1 [, join\_column\_2]...)]... |
| --- |

### NATURAL JOIN

* Natural join - joins two tables together based on all columns in the two tables having the same name

| SELECT select\_list  FROM base\_table  NATURAL JOIN table\_1  [NATURAL JOIN table\_2] … |
| --- |

### CROSS JOIN

* Cross join - joins each row from the first table with each row from the second table
  + Cartesian product - the result set

| SELECT select\_list  FROM table\_1 CROSS JOIN table\_2 |
| --- |

### UNION

* Union - combines data from two or more tables
  + UNION - to connect two or more SELECT statements, the SELECT statements must have the same number of columns and the data types of the corresponding columns in each table must be compatible

| SELECT\_statement\_1  UNION [ALL]  SELECT\_statement\_2  [UNION [ALL]  {SELECT\_statement\_3] … |
| --- |

* Full outer join - the result set includes all the rows from both tables

## Chapter 5: How to Insert, Update, and Delete Data

### CREATE and DROP TABLE

* The result set that’s defined by the SELECT statement is copied into a new table.
  + MySQL only copies the column definitions and data. Doesn’t retain primary keys, foreign keys, and indexes

| CREATE TABLE table\_name AS select\_statement |
| --- |

* Drop Table - Deletes any tables you don’t need anymore

| DROP TABLE table\_name |
| --- |

### INSERT

* Insert - to add rows to the table
  + You name the table followed by an optional list of columns.
  + You are to list the values to be inserted in the VALUES clause. If you don’t include a column list in the INSERT clause, you must specify the column values in the same order as in the table, and you must code a value for each column. Each row after the VALUES keyword affects a row

| INSERT [INTO] table\_name  [(column\_list)]  VALUES  (expression\_1[, expression\_2]...)[,  (expression\_1[, expression\_2}...)]... |
| --- |

* + Insert default values and null values
    - If a column is defined so it allows null values, you can use the NULL keyword in the list of values to insert a null value into that column
    - If a column is defined with a default value, you can use the DEFAULT keyword in the list of values to insert the default value for that column
    - If a column is defined as an auto-increment column, you can use the DEFAULT keyword in the list of values to have MySQL generate value for the column
    - Column example: color\_sample

| color\_id | INT | NOT NULL | AUTO\_INCREMENT, |
| --- | --- | --- | --- |
| color\_number | INT | NOT NULL | DEFAULT 0, |
| color\_name | VARCHAR(50) |  |  |

* + Using a subquery in an INSERT statement
    - Subquery - just a SELECT statement that’s coded within another SQL statement.

| INSERT [INTO] table\_name  [(column\_list)]  select\_statement |
| --- |

Example

| INSERT INTO invoice\_archive  SELECT \*  FROM invoices  WHERE invoice\_total - payment\_total - credit\_total = 0 |
| --- |

### UPDATE

* You use the UPDATE statement to modify the data in one or more rows of a table
  + The SET clause names the columns to be updated and the values to be assigned to those columns
  + WHERE clause specifies the condition a row must meet in order to be updated
  + You can use the DEFAULT and NULL keywords to specify default and null values

| UPDATE table\_name  SET column\_name\_1 = expression\_1[, column\_name\_2 = expression\_2] …  [WHERE search\_condition] |
| --- |

* You can use a subquery in the WHERE clause

| UPDATE table\_name  SET column\_name\_1 = expression\_1[, column\_name\_2 = expression\_2] …  WHERE column\_name\_2 IN  (SELECT column\_name\_2  FROM table  WHERE search\_condition) |
| --- |

### DELETE FROM

* You can use the DELETE statement to delete one or more rows from the table you name in the DELETE clause
  + Must include the FROM clause to get the table and WHERE clause with a search condition

| DELETE FROM table\_name  WHERE search\_condition |
| --- |

* You can also use a subquery in the WHERE clause

| DELETE FROM table\_name  WHERE column\_name\_2 IN  (SELECT column\_name\_2  FROM table  WHERE search\_condition) |
| --- |

# MORE SQL skills as you need them

## Chapter 6: How to code summary queries

### Aggregate functions

* Aggregate functions - column functions - perform a calculation on the values in a set of selected rows
* Summary query - is a SELECT statement that includes one or more aggregate functions

| Function syntax | Result |
| --- | --- |
| AVG([ALL|DISTINCT] expression) | Average of the non-null values in teh expression |
| SUM([ALL|DISTINCT] expression) | The total of the non-null values in the expression |
| MIN([ALL|DISTINCT] expression) | The lowest non-null value in the expression |
| MAX([ALL|DISTINCT] expression) | The highest non-null value in the expression |
| COUNT([ALL|DISTINCT] expression) | The number of non-null values in the expression |
| COUNT(\*) | The number of rows selected by the query |

| SELECT select\_list  Aggregate\_function([ALL|DISTINCT] expression)  FROM table  WHERE search\_condition |
| --- |

* + To count only the selected rows with unique values in a specified column, you can code the COUNT function with the DISTINCT keyword followed by the name of the column

### GROUP BY

* GROUP BY - groups the rows of a result set based on one or more columns or expressions.
  + To include two or more columns or expressions, separate them by commas. They form a hierarchy where each column or expression is subordinate to the previous one
  + You can include aggregate functions in the SELECT clause, the aggregate is calculated for each group specified by the GROUP BY clause

### HAVING

* HAVING - specifies a search condition for a group or an aggregate. MySQL applies this condition after it groups the rows that satisfy the search condition in the WHERE clause
  + You can have AND and OR operators to code compound search conditions
  + When you include a HAVING clause in a SELECT statement that uses grouping and aggregate functions, MySQL applied the search condition after it groups the rows and calculates the aggregates. WHERE on the other hand applies the search condition before it groups the rows and calculates the aggregates
    - WHERE can refer to any column in the base tables and can’t contain any aggregate functions
    - HAVING can only refer to a column included in the SELECT statement and can contain aggregate functions
  + The HAVING clause can have compound conditions
  + You have GROUP BY programmed without HAVING but you can’t have HAVING without GROUP BY
* Functionally dependent - a relationship that exists when one attribute uniquely determines another attribute

| SELECT select\_list  FROM table  [WHERE search\_condition]  [GROUP BY group\_by\_list]  [HAVING search\_condition]  [ORDER BY order\_by\_list] |
| --- |

### WITH ROLLUP

* WITH ROLLUP - you can use this operator in the GROUP BY clause to add summary rows to the final result set
  + Adds a summary to each group specified in the GROUP BY clause. It also adds a summary row to the end of the result set that summarizes the entire result set

| SELECT select\_list  FROM table  WHERE search\_condition  GROUP BY group\_by\_list WITH ROLLUP |
| --- |

### GROUPING

* GROUPING - a function that returns a 1 if the expression is null because it’s in a summary row. Otherwise, it returns a 0
  + This is commonly used to replace the nulls that are generated by WITH ROLLUP with literal values.

| SELECT IF(GROUPING(column\_name\_1) = 1, ‘replacement\_name’, column\_name\_1) AS table\_alias, [IF(GROUPING(column\_name\_2) = 1, ‘replacement\_name’, column\_name\_2) AS table\_alias] …  FROM table  WHERE search\_condition  GROUP BY group\_by\_list WITH ROLLUP |
| --- |

### Other stuff

* Aggregate window functions - operate on a set of rows and return a single value for each row from the underlying query. To code this you use the OVER clause.
* OVER - this defines the window that’s used by the aggregate function.
  + A window consists of all the rows that are needed to evaluate the function for the current row
* PARTITION BY - partitions the result set by the vendor\_column
  + If you code an OVER clause with a PARTITION BY clause, the aggregate function is performed on each partition

OVER clause syntax

| OVER ([PARTITION BY expression1 [, expression2] …  [ORDER BY expression1 [ASC|DESC] [, expression2 [ASC|DESC]]...) |
| --- |

Example

| SELECT vendor\_id, invoice\_date, invoice\_total,  SUM(invoice\_total) OVER() AS total\_invoices,  SUM(invoice\_total) OVER(PARTITION BY vendor\_id) AS vendor\_total  FROM invoices  WHERE invoice\_total > 5000; |
| --- |

* If you code an ORDER BY clause on the OVER clause, the rows within each partition are sorted and the values from one row to the next are cumulative

Example

| SELECT vendor\_id, invoice\_date, invoice\_total,  SUM(invoice\_total) OVER() AS total\_invoices,  SUM(invoice\_total) OVER(PARTITION BY vendor\_id ORDER BY invoice\_total) AS vendor\_total  FROM invoices  WHERE invoice\_total > 5000; |
| --- |

* Frame - the number of rows before and after the current row (ROWS) or a range of values based on the value of the current row (RANGE)
  + To specify the starting and ending row, you use the BETWEEN clause

Frame syntax

| {ROWS | RANGE} {frame\_start | BETWEEN frame\_start AND frame\_end} |
| --- |

Possible values for frame\_start and frame\_end

| Value | Description |
| --- | --- |
| CURRENT ROW | The frame starts and and withe current row |
| UNBOUNDED PRECEDING | The frame starts and ends with the first row in the partition |
| UNBOUNDED FOLLOWING | The frame starts and ends with the last row in the partition |
| expr PRECEDING | With ROWS, the frame starts expr rows before the current row. With RANGE, the frame starts with the first row before the current row whose value is expr less than the value of the current row. |
| expr FOLLOWING | With ROWS, the frame starts expr rows after the current row. With RANGE, the frame starts with the last row after the current row whose value is expr greater than the value of the current row. |

Example

| SELECT vendor\_id, invoice\_date, invoice\_total,  SUM(invoice\_total) OVER() AS total\_invoices,  SUM(invoice\_total) OVER(PARTITION BY vendor\_id ORDER BY invoice\_date  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS vendor\_total  FROM invoices  WHERE invoice\_date BETWEEN '2018-04-01' AND '2018-04-30'; |
| --- |

* If you code an ORDER BY clause on the OVER clause and you use the RANGE keyword, values are accumulated up to and including the current row as well as its peers
  + Peer - a row that’s in the same sort sequence as other rows in the partition
  + You can use a frame to calculate a moving average. A moving average is calculated by adding the value of the current row to the values of zero or more preceding and following rows

Example

| SELECT MONTH(invoice\_date) AS month,  SUM(invoice\_total) AS total\_invoices,  ROUND(AVG(SUM(invoice\_total)) OVER(ORDER BY MONTH(invoice\_date)  RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING), 2) AS 3\_month\_avg  FROM invoices  GROUP BY MONTH(invoice\_date); |
| --- |

* Named window - in some cases you’ll need to code a SELECT statement with two or more aggregate functions that use the same window. You use this so you don’t have to repeat the definition for the window in each function
  + You use the WINDOW clause. This is placed after HAVING and before ORDER BY
  + To use a named window, you code it on the OVER clause. If you code just the window name, you don’t include parentheses.

Window syntax

| WINDOW window\_name AS ([partition\_clause] [order\_clause] [frame\_clause]) |
| --- |

Example

| SELECT vendor\_id, invoice\_date, invoice\_total,  SUM(invoice\_total) OVER vendor\_window AS vendor\_total,  ROUND(AVG(invoice\_total) OVER vendor\_window, 2) AS vendor\_avg,  MAX(invoice\_total) OVER vendor\_window AS vendor\_max,  MIN(invoice\_total) OVER vendor\_window AS vendor\_min  FROM invoices  WHERE invoice\_total > 5000  WINDOW vendor\_window AS (PARTITION BY vendor\_id); |
| --- |

## Chapter 7: How to code subqueries

### Introduction to subqueries in a select statement

* Subquery - a a SELECT statement that’s coded within another SQL statement
  + This can be introduced in the WHERE, HAVING, FROM, or SELECT clause of a SELECT statement
  + It can return a single value, a list of values, or a table of values
  + The subquery can’t include an ORDER BY

| SELECT invoice\_number, invoice\_date, invoice\_total  FROM invoices  WHERE invoice\_total >  (SELECT AVG(invoice\_total)  FROM invoices)  ORDER BY invoice\_total; |
| --- |

* When to use subqueries - most subqueiries can be restated as joins
  + Advantages of joins
    - The SELECT clause of a join can include columns from both tables
    - A join tends to be more intuitive when it uses an existing relationship between the two tables, such as a primary key to foreign key relationship
  + Advantages of subqueries
    - You can use a subquery to pass an aggregate value to the main query
    - A subquery tends to be more intuitive when it uses an ad hoc relationship between the two tables
    - Long, complex queries can sometimes be easier to code using subqueries

### Subqueries in WHERE clause

* + You can introduce a subquery with the IN operator to provide the list of values that are tested against the test expression
  + When you use the IN operator, the subquery must return a single column of values
  + A query that uses the NOT IN operator with a subquery can typically be restated using an outer join

| WHERE test\_expression [NOT] IN (subquery) |
| --- |

* Comparison operators
  + If you code a search condition with ANY, SOME, or ALL keyword, the subquery can return a list of values. Without it the subquery must return a single value.

| WHERE expression comparison\_operator [SOME|ANY|ALL] (subquery) |
| --- |

* + ALL - returns a boolean value as a result. Returns TRUE if ALL of the subquery values meet the condition. Is used with SELECT, WHERE and HAVING statements

| Condition | Equivalent Expression | Description |
| --- | --- | --- |
| X > ALL (1, 2) | X > 2 | Evaluate to true if x is greater than the maximum value returned by the subquery |
| X < ALL (1, 2) | X < 1 | Evaluates to true if x is less than the minimum value returned by the subquery |
| X = ALL (1, 2) | (X = 1) AND (X = 2) | Evaluates to trueif the subquery returns a single value that’s equal to x or if the subquery returns multiple values that are all equal to x |
| X <> ALL (1, 2) | X NOT IN (1, 2) | Evaluates to true if x is not one of the value returned by the subquery |

* + ANY - Used to test that a condition is true for one or more of the values returned by a subquery. If the subquery doesn’t return any values, or if it only returns null values, a comparison that uses the ANY keyword evaluates to false.
    - SOME keyword works the same as ANY

| Condition | Equivalent Expression | Description |
| --- | --- | --- |
| X > ANY (1, 2) | X > 1 | Evaluates to true if X is greater than the minimum value returned by the subquery |
| X < ANY (1, 2) | X < 2 | Evaluates to true if X is less than the maximum value returned by the subquery |
| X = ANY (1, 2) | X IN (1, 2) | Evaluates to true if X is equal to any of the values returned by the subquery |
| X <> ANY (1, 2) | (X <> 1) OR (X <> 2) | Evaluates to true if X is not equal to at least one of the values returned by the subquery |

* Correlated Subquery - a subquery that is executed once for each row in the main query
  + Refers to a value that’s provided by a column in the main query. For each different value that’s returned by the main query for that column, the subquery returns a different result
* Uncorrelated Subquery - is executed only once
* EXISTS - tests whether the subquery returns a result set
  + You can use this operator to test that the subquery returns one or more rows
  + You can use the NOT EXISTS operator to test that no rows are returned by the subquery
  + When you use these operators with a subquery, it doesn’t matter what columns you specify in the SELECT clause. As a result, you typically just code an \*

| WHERE [NOT] EXISTS (subquery) |
| --- |

* + When you code a HAVING clause, you specify a search condition just as you do when you code a WHERE clause. That includes having conditions that contain subqueries.
  + When you code a subquery in a SELECT clause, the subquery must return a single value
  + When you code a subquery in a SELECT clause, you typically use a correlated subquery.
* Example

| SELECT vendor\_name,  (SELECT MAX(invoice\_date) FROM invoices  WHERE vendor\_id = vendors.vendor\_id) AS latest\_inv  FROM vendors  ORDER BY latest\_inv DESC; |
| --- |

### Coding subqueries in a FROM clause

* Returns a result set that can be referred to as an inline view
* You must assign an alias to it, then you can use that alias just as you would any other table name
  + You should use an alias for any columns in the subquery that perform calculations
* Example

| SELECT vendor\_state, MAX(sum\_of\_invoices) AS max\_sum\_of\_invoices  FROM  (  SELECT vendor\_state, vendor\_name,  SUM(invoice\_total) AS sum\_of\_invoices  FROM vendors v JOIN invoices i  ON v.vendor\_id = i.vendor\_id  GROUP BY vendor\_state, vendor\_name  ) t  GROUP BY vendor\_state  ORDER BY vendor\_state; |
| --- |

### How to work with complex queries

| SELECT t1.vendor\_state, vendor\_name, t1.sum\_of\_invoices  FROM  (  -- invoice totals by vendor  SELECT vendor\_state, vendor\_name,  SUM(invoice\_total) AS sum\_of\_invoices  FROM vendors v JOIN invoices i  ON v.vendor\_id = i.vendor\_id  GROUP BY vendor\_state, vendor\_name  ) t1  JOIN  (  -- top invoice totals by state  SELECT vendor\_state,  MAX(sum\_of\_invoices) AS sum\_of\_invoices  FROM  (  -- invoice totals by vendor  SELECT vendor\_state, vendor\_name,  SUM(invoice\_total) AS sum\_of\_invoices  FROM vendors v JOIN invoices i  ON v.vendor\_id = i.vendor\_id  GROUP BY vendor\_state, vendor\_name  ) t2  GROUP BY vendor\_state  ) t3  ON t1.vendor\_state = t3.vendor\_state AND  t1.sum\_of\_invoices = t3.sum\_of\_invoices  ORDER BY vendor\_state; |
| --- |

1. This query retrieves the vendor from each state that has the largest invoice total. To do that, it uses 3 subqueries
2. The subqueries named t1 and t2 return the same result set. This result set includes the vendor state, name, and sum of invoices
3. The subquery named t3 returns a result set that includes the vendor state and the largest sum of invoices for any vendor in that state. To do that, this subquery uses a nested subquery named t2
4. The subqueries named t1 and t3 are joined on both the vendor\_state and sum\_of\_invoices columns.

### Procedure for building complex queries

1. State the problem to be solved by the query in English
2. Use pseudocode to outline the query
3. Code the subqueries and test them to be sure that they return the correct data
4. Code and test the final query

### Common table expressions

* Common table expression (CTE) is a new feature of MySQL 8.0 that allows you to code an expression that defines a named temporary result set that can be used by the query that follows
  + To use a CTE, you code the WITH keyword followed by the definition of the CTE.
    - After you code the CTE you code the statement that uses it
  + To code multiple CTEs, separate them with commas. Then each CTE can refer to itself and any previously defined CTEs with the same WITH clause
  + You will most likely use CTEs in SELECT statements

| WITH [RECURSIVE] cte\_name1 AS (subquery)  [, cte\_name2 AS (subquery2)]  [...]  sql\_statement |
| --- |

Example

| WITH summary AS  (  SELECT vendor\_state, vendor\_name, SUM(invoice\_total) AS sum\_of\_invoices  FROM vendors v JOIN invoices i  ON v.vendor\_id = i.vendor\_id  GROUP BY vendor\_state, vendor\_name  ),  top\_in\_state AS  (  SELECT vendor\_state, MAX(sum\_of\_invoices) AS sum\_of\_invoices  FROM summary  GROUP BY vendor\_state  )  SELECT summary.vendor\_state, summary.vendor\_name, top\_in\_state.sum\_of\_invoices  FROM summary JOIN top\_in\_state  ON summary.vendor\_state = top\_in\_state.vendor\_state AND  summary.sum\_of\_invoices = top\_in\_state.sum\_of\_invoices  ORDER BY summary.vendor\_state; |
| --- |

* A recursive query is a query that can loop through a result set and perform processing to return a final set. (A recursive CTE can be used to create this)

| WITH RECURSIVE employees\_cte AS  (  -- Anchor member  SELECT employee\_id,  CONCAT(first\_name, ' ', last\_name) AS employee\_name,  1 AS ranking  FROM employees  WHERE manager\_id IS NULL  UNION ALL  -- Recursive member  SELECT employees.employee\_id,  CONCAT(first\_name, ' ', last\_name),  ranking + 1  FROM employees  JOIN employees\_cte  ON employees.manager\_id = employees\_cte.employee\_id  )  SELECT \*  FROM employees\_cte  ORDER BY ranking, employee\_id; |
| --- |

## Chapter 8: How to work with data types

### Data Types

* Character - strings of character data
  + Type - Bytes
    - CHAR(M) - Mx4 - M number of characters
    - VARCHAR(M) - L+1 - L length of string plus 1
* Numeric - Numbers that don’t include a decimal point (integers) and numbers that include a decimal point (real numbers)
  + Type - Bytes
    - BIGINT - 8
    - INT - 4
    - MEDIUMINT - 3
    - SMALLINT - 2
    - TINYINT - 1
  + Fixed-point type
    - DECIMAL(M, D) - Vary - M is maximum number of total digits and D is the number of digits to the right of the decimal
  + Floating-point types
    - DOUBLE - 8
    - FLOAT - 4
* Date and Time - dates, times, or both
  + Type - Bytes
    - DATE - 3
    - TIME - 3
    - DATETIME - 8 - defaults to “yyyy-mm-dd hh:mm:ss”
    - TIMESTAMP - 4
    - YEAR[(4)] - 1
* ENUM and SET - character data types since they allow you to restrict the values for a column to a limited set of strings
  + Type - Bytes
    - ENUM - 1-2 - stores one value from a list of acceptable values
    - SET - 1-8 - stores 0 or more values selected from a list of acceptable values
* Large Object (LOB) - large string of character or binary data (images, sound, video, large amount of text)
  + Type - Bytes
    - LONGBLOB L+4
    - MEDIUMBLOB L+3
    - BLOB L+2
    - TINYBLOB L+1
    - LONGTEXT L+4
    - MEDIUMTEXT L+3
    - TEXT L+2
    - TINYTEXT L+1
  + BLOB (binary large object) types
  + CLOB (character large object) types
* Spatial - geographical values
* JSON - JSON documents

### How to convert data

* Implicit conversion - MySQL automatically converts one data type to another
  + Number to string
    - CONCAT
  + String to number
  + Date to number
    - Add a numeric value to it

| CAST(expression AS cast\_type) |
| --- |

| CONVERT(expression, cast\_type) |
| --- |

* Cast types you can use
  + CHAR[(N)]
  + DATE
  + DATETIME
  + TIME
  + SIGNED [INTEGER]
  + UNSIGNED [INTEGER]
  + DECIMAL[(M[, D])]
* FORMAT - converts the specified number to a character string with grouped digits separated by commas, rounded to the specified number of decimal digits. If the decimal is zero, then the decimal point is omitted

| FORMAT(number, decimal) |
| --- |

* CHAR - converts one or more numbers to a binary string. Each number is interpreted as an integer between 0 and 255

| CHAR(value1[,value2]...) |
| --- |

## Chapter 9: How to use functions

### How to work with string data

* CONCAT(str1[,str2]...) - Adds two or more expressions together
* CONCAT\_WS(sep,str1[,str2]...) - Adds two or more expressions together with a separator
* LTRIM(str) - Removes leading spaces from a string
* RTRIM(str) - Removes trailing spaces from a string
* TRIM([[BOTH|LEADING|TRAILING] [remove] FROM] str) - Removes leading and trailing spaces from a string
* LENGTH(str) - Returns the length of a string (in bytes)
* LOCATE(find,search[,start]) - Returns the position of the first occurrence of a substring in a string
* LEFT(str,length) - Extracts a number of characters from a string (starting from left)
* RIGHT(str,length) - Extracts a number of characters from a string (starting from right)
* SUBSTRING\_INDEX(str,delimiter, count) - Returns a substring of a string before a specified number of delimiter occurs
* SUBSTRING(str,start[,length]) - Extracts a substring from a string (starting at any position)
* REPLACE(search,find,replace) - Replaces all occurrences of a substring within a string, with a new substring
* INSERT(str,start,length,insert) - Inserts a string within a string at the specified position and for a certain number of characters
* REVERSE(str) - Reverses a string and returns the result
* LOWER(str) - Converts a string to lower-case
* UPPER(str) - Converts a string to upper-case
* LPAD(str,length,pad) - Left-pads a string with another string, to a certain length
* RPAD(str,length,pad) - Right-pads a string with another string, to a certain length
* SPACE(count) - Returns a string of the specified number of space characters
* REPEAT(str,count) - Repeats a string as many times as specified

### Some of the numeric functions

* ROUND(number[,length]) - Rounds a number to a specified number of decimal places
* TRUNCATE(number,length) - Truncates a number to the specified number of decimal places
* CEILING(number) - Returns the smallest integer value that is >= to a number
* FLOOR(number) - Returns the largest integer value that is <= to a number
* ABS(number) - Returns the absolute value of a number
* SIGN(number) - Returns the sign of a number
* SQRT(number) - Returns the square root of a number
* POWER(number,power) - Returns the value of a number raised to the power of another number
* RAND([integer]) - Returns a random number

### Functions that get the current date and time

* NOW()
* SYSDATE()
* CURRENT\_TIMESTAMP()
* CURDATE()
* CURRENT\_DATE()
* CURTIME()
* CURRENT\_TIME()
* UTC\_DATE()
* UTC\_TIME()

### How to parse dates and times with date/time functions

* DAYOFMONTH(date) - Returns the day of the month for a given date
* MONTH(date) - Returns the month part for a given date
* YEAR(date) - Returns the year part for a given date
* HOUR(time) - Returns the hour part for a given date
* MINUTE(time) - Returns the minute part of a time/datetime
* SECOND(time) - Returns the seconds part of a time/datetime
* DAYOFWEEK(date) - Returns the weekday index for a given date
* QUARTER(date) - Returns the quarter of the year for a given date value
* DAYOFYEAR(date) - Returns the day of the year for a given date
* WEEK(date[,first]) - Returns the week number for a given date
* LAST\_DAY(date) - Extracts the last day of the month for a given date
* DAYNAME(date) - Returns the weekday name for a given date
* MONTHNAME(date) - Returns the name of the month for a given date

### How to format dates and times

* DATE\_FORMAT(date,format)
* TIME\_FORMAT(time,format)
* Code - Description
  + %m - Month, numeric (01...12)
  + %c - Month, numeric (1...12)
  + %M - Month name (January...December)
  + %b - Abbreviated month name (Jan...Dec)
  + %d - Day of the month, numeric (00...31)
  + %e - Day of the month, numeric (0...31)
  + %D - Day of the month with suffix (1st, 2nd, 3rd, etc.)
  + %y - Year, numeric, 2 digits
  + %Y - Year, numeric, 4 digits
  + %W - Weekday name (Sunday...Saturday)
  + %a - Abbreviated weekday name (Sun...Sat)
  + %H - Hour (00...23)
  + %k - Hour (0...23)
  + %h - Hour (01...12)
  + %l - Hour (1...12)
  + %i - Minutes (00...59)
  + %r - Time, 12-hour (hh:mm:ss AM or PM)
  + %T - Time, 24-hour (hh:mm:ss)
  + %S - Seconds (00...59)
  + %p - AM or PM

### How to perform calculations on dates and times

* DATE\_ADD(date,INTERVAL expression unit)
* DATE\_SUB(date,INTERVAL expression unit)
* DATEDIFF(date1, date2)
* TO\_DAYS(date)
* TIME\_TO\_SEC(time)

### Other functions

* Simple Case function

| CASE input\_expression  WHEN when\_expression\_1 THEN result\_expression\_1  [WHEN when\_expression\_2 THEN result\_expression\_2]...  [ELSE else\_result\_expression]  END |
| --- |

* Searched Case function

| CASE  WHEN conditional\_expression\_1  THEN result\_expression\_1  [WHEN conditional\_expression\_2  THEN result\_expression\_2]...  [ELSE else\_result\_expression]  END |
| --- |

* IF(test\_expression, if\_true\_expression, else\_expression)
* IFNULL(test\_expression, replacement\_value)
* COALESCE(expression\_1[, expression\_2]...)

### How to use the regular expression functions

* REGEXP\_LIKE(expr, pattern)
* REGEXP\_INSTR(expr, pattern [, start])
* REGEXP\_SUBSTR(expr, pattern [, start])
* REGEXP\_REPLACE(expr, pattern, replace[, start])
* Character/Construct - Description
  + ^ - Matches the pattern to the beginning of the value.
  + $ - Matches the pattern to the end of the value.
  + . - Matches any single character.
  + [charlist] - Matches any single character listed within the brackets.
  + [char1–char2] - Matches any single character within the given range.
  + | - Separates two string patterns and matches either one.
  + char\* - Matches zero or more occurrences of the character.
  + (charlist)\* - Matches zero or more occurrences of the sequence of characters in parentheses.

### How to use the ranking functions

* ROW\_NUMBER() OVER([partition\_clause] order\_clause)
* RANK() OVER([partition\_clause] order\_clause)
* DENSE\_RANK() OVER([partition\_clause] order\_clause)
* NTILE(integer\_expression) OVER([partition\_clause] order\_clause)

### How to use the analytic functions

| {FIRST\_VALUE|LAST\_VALUE|NTH\_VALUE}  (scalar\_expression[, numeric\_literal])  OVER ([partition\_clause] order\_clause [frame\_clause])  {LEAD|LAG}(scalar\_expression [, offset [, default]])  OVER ([partition\_clause] order\_clause)  {PERCENT\_RANK()|CUME\_DIST()}  OVER ([partition\_clause] order\_clause) |
| --- |

# Database design and implementation

## Chapter 10: How to design a database

### Basic steps

* The six basic steps for designing a data structure
  + Identify the data elements
  + Subdivide each element into its smallest useful components
  + Identify the tables and assign columns
  + Identify the primary and foreign keys
  + Review whether the data structure is normalized
  + Identity the indexes

### What they should look like

* A relational database system should model real-world environment where it is used
* A table in a relational database typically represents an object, or entity, in the real world.
  + Each column of a table is used to store an attribute associated with the entity
  + Each row represents one instance of the entity
  + Entity-relationship modeling - modeling a database and relationships between its tables after a real-world system
* Each table should have a primary key that uniquely identifies each row
  + The primary keys should rarely change
* If 2 tables have a one-to-many relationship, you may need to add a foreign key column to the table on the “many” side.
  + The foreign key must have the same datatype as the primary key
* If 2 tables have a many-to-many relationship, you’ll need to define a linking table to relate them.
* If 2 tables have a one-to-one relationship, they should be related by their primary keys.
* There are Operations that can violate referential integrity
  + Deleting a row from the primary key table
    - If the foreign key table contains one or more rows related to the deleted row
  + Inserting a row in the foreign key table
    - If the foreign key value doesn’t have a matching primary key value in the related table
  + Updating the value of a foreign key
    - If the new foreign key value doesn’t have a matching primary key value in the related table
  + Updating the value of a primary key
    - If the foreign key table contains one or more rows related to the row that’s changed

### Referential integrity

* Referential integrity - the relationships between tables are maintained correctly. A table with a foreign key doesn’t have rows with foreign key values that don’t have matching primary key values in the related table
  + Declarative referential integrity - (DRI) ensures the integrity of the database by a properly managed primary key and foreign key relationship.
  + If referential integrity isn’t enforced and a row is deleted from the primary key table that has related rows in the foreign key table, the rows in the foreign key table are said to be orphaned

### Normalization

* Normalization - the process of organizing data in a database
  + More tables, and each table has an index on its primary key. That makes data retrieval more efficient.
  + Each table contains information about a single entity. That makes data retrieval and insert, update, and delete operations more efficient.
  + Each table has fewer indexes, which makes insert, update, and delete operations more efficient.
  + Data redundancy is minimized, which simplifies maintenance and reduces storage.
  + There are 7 forms of normalization but a data structure is typically considered normalization if the first three normal forms are applied:
    - First (1NF) The value stored at the intersection of each row and column must be a scalar value, and a table must not contain any repeating columns.
    - Second (2NF) Every non-key column must depend on the primary key.
    - Third (3NF) Every non-key column must depend only on the primary key.
    - Boyce-Codd (BCNF) A non-key column can’t depend on another non-key column.
    - Fourth (4NF) A table must not have more than one multivalued dependency, where the primary key has a one-to-many relationship to non-key columns.
    - Fifth (5NF) The data structure is split into smaller and smaller tables until all redundancy has been eliminated.
    - Domain-key (DKNF) or Sixth (6NF) Every constraint on the relationship is dependent only on key constraints and domain constraints, where a domain is the set of allowable values for a column.

### When to create an index:

* When the column is used frequently in search conditions or joins?
* When the column contains a large number of distinct values?
* When the column is updated infrequently?

### When to denormalize

* When a column from a joined table is used repeatedly in search criteria.
* If a table is updated infrequently.
* Include columns with derived values when those values are used frequently in search conditions.