SQL

Most common commands:
SELECT - extracts data from a database
UPDATE - updates data in a database
DELETE - deletes data from a database
INSERT INTO - inserts new data into a database
CREATE DATABASE - creates a new database
ALTER DATABASE - modifies a database
CREATE TABLE - creates a new table
ALTER TABLE - modifies a table
DROP TABLE - deletes a table
CREATE INDEX - creates an index (search key)
DROP INDEX - deletes an index

exampe 1
`SELECT DISTINCT Country FROM Customers;` - **Avoid duplicated values**

example 2 - There is employees table, and each row has city. List the number of different employees cities:
My first attempt was:
`select count(*) from (select count(*) from employees group by city) mustHaveAlias`
How to do it:
`SELECT COUNT(DISTINCT city) FROM employees;`

-- Syntax for SQL Server and Azure SQL Database

<search_condition> ::=
{ [NOT] | (<search_condition>) }
[{ AND | OR } [NOT] { | (<search_condition>) }]
[,...n]

::=
{ expression { = | < > | ! = | > | > = | ! > | < | < = | ! < } expression
| string_expression [NOT] LIKE string_expression
[ESCAPE 'escape_character']
| expression [NOT] BETWEEN expression AND expression
| expression IS [NOT] NULL
| CONTAINS
({ column | * } , '<contains_search_condition>')
| FREETEXT ({ column | * } , 'freetext_string')
| expression [NOT] IN (subquery | expression [,...n])
| expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
{ ALL | SOME | ANY} (subquery)
| EXISTS (subquery) }

Order by
//TODO

Insert into

> INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
> `insert into employees (city, name) values ('Test City', 'MILENI')` - order of the columns can
> be changed. Columns can be ommited if we insert all columns.

NULL Value - when left blank during record creation. ...where column_name IS (NOT) NULL

Update

```
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Delete

```
DELETE FROM table_name
WHERE condition;
```

Min, Max, count, avg, sum

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

JOINS - INNER, LEFT, RIGHT, FULL
Join **clause** - to combine rows from two or more tables. TableA, TableB - **NOTE** for each type of join, rows from both tables are combined together (each-with-each)???

- INNER - Selects records from TableA and TableB where join condition is met.
- LEFT - Selects all records from TableA, along with records from TableB for which condition is met.
- RIGHT - Selects all records from TableB, along with records from TalbeA for which condition is met.
- FUL - Selects all reconds from TalbeA and TableB regardless of whether conditions are met or not.
- *self* - combined with itself

Join examples

`select * from tableA, tableB` - like `select * from tableA inner join tableB (on...)`
`select * from tableA left join tableB on 1 = 1` - same as ^^ since tableB rows always meet given condition.
`select * from employees left join orders on employees.id = orders.epmployee_id` This simply prepends order information on employees table rows. If there is no matching order, fields will be NULL.

example 2 - there are two tables employees and orders;
left join - all rows from left table will appear at least once. right join -||-

- select all employees and order info no matter if they have made order or not:
```select * from employees left join orders on employees.id = orders.employee_id``
- select all users with orders (if each order is attached to some users, there won't be null columns):
```select * from employees right join orders on employees.id = orders.employee_id``

UNION - To combine resultSets from multiple select statements (they all should have same number of columns)
UNION ALL - to support duplicated items.

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Group by - this statement is **often** used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns. I can even write:
`select * from employees group by city` - If there are multiple rows with the same city, first one will be taken. => it groups all rows into one where key is city (values are overridden)

example 1 - count each city in employees table:
attempt: `select city, count(city) from employees` - this gives you first city that appears and total count of rows => it needs `group by` at the end.
general rule - every column that appears in select should appear in group by as well.

Having - like **where** but applies after aggregate function.
Example: Select cities that appear at lest twice in employees table.
`select city, count(*) as num from employees group by city having num >= 2`

Exists

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

Example - select employees that have at least one order:
`select * from employees e where exists (select * from orders where orders.employee_id = e.id)`

SQLite

//TODO

T-SQL

t-sql on tutorialspoint

More examples

example 1
`select * from employees where name in ('Milos', 'Marko', 'Stanko')`

example 2 - **Variables** - must start with @ otherwise it's treated as system variable.

```
set @user = 'Milos';
select @user
```

example 3 select into variable:
`select @maxSalary:=max(salary) from employees`

Transactions in *mysql*
set autocommit = 0 - to disable autocommit, otherwise changes are persistend on every update and can't rollback.

```
start transaction; #automatically sets autocommit to 0
select @A:=sum(salary) from table1 where type=1;
update table2 set summary=@A where type=1;
commit;
```

If I try to update, then select to check changes, I will see updated table event before calling **commit**
but everything can be canceled with **rollback**. When commit, can't rollback anymore.

Normalization

- Normalization is a technique of organizing data into multiple related tables to minimize **data redundancy**
- insertion anomaly - duplication data on insertion, sometimes maybe multiple rows
- updation anomaly - If each student holds branch info, like mailing list, and mailing list changes, we need to
  update every single row in students table.
- deletion anomaly - There is a students table that holds branch data as well (no branch table at all). If we delete all students, then all branch data will be lost.
- **1NF** (First Normal form). 4 rules:
  1. single (atomic) data in columns (it depends how we treat data, for instance string is atomic but if we keep
     CSV like "Java, C++" then it's not atomic.
  2. values in the same column should be of the same domain(don't put name in 'created_date' column)
  3. all the columns have unique names
  4. the order in which data is stored doesn't matter
- **2NF**. 2 Rules:

1. Must be 1NF
2. Should not have **partial dependency**
   Functional dependency - all I need is *student_id* (primary key) and every other column depends on it,
   or can be fetched using it.
   Partial dependency - let's start with R{ABCD}, and the functional dependencies AB->CD and A->C.
   The only candidate key for R is AB. C and D are a nonprime attributes. C is functionally dependent on A.
   A is part of a candidate key. That's a partial dependency.
   A relation is said to be in 2NF if and only if it is in 1NF and every non key attribute is fully dependent on the primary key. After 2NF, we can still have redundant data.