Introduction to SQL:
- SQL (Structured Query Language) is a formal language for doing so with a DBMS.
- There are 2 parts to SQL:
    1. **DDL (Data Definition Language):** Used for defining schemas.
    2. **DML (Data Manipulation Language):** Used for writing queries and modifying the database.
    I.e. Whenever we are defining schemas, it's called DDL and whenever we are writing queries, it's called DML.
- SQL is a very high-level language.
- It provides **physical data independence** meaning that details of how the data is stored can change with no impact on your queries.
- You can focus on readability and because the DBMS optimizes your query, you get efficiency.
- SQL keywords/commands are not case sensitive.
    I.e. select = SELECT
    One convention is to use uppercase for keywords.
- Identifiers are not case-sensitive either.
    One convention is to use lowercase for attributes, and a leading capital letter followed by lowercase for relations.
- Literal strings are case-sensitive, and require single quotes.
- Whitespace (other than inside quotes) is ignored.
- E.g. Consider the query **SELECT surName FROM Student WHERE campus = 'StG';**
    SELECT, FROM and WHERE, which are all keywords, are all in capital letters.
    surName, and campus, which are identifiers for attributes, are lowercase.
    Student, which is an identifier for a relation, starts with an uppercase letter but everything else is lowercase.
    'StG', which is a literal string, is case sensitive and must be surrounded by single quotes.
- There are SQL statements, clauses, operators and functions. I will list the SQL statements in alphabetical order, followed by the SQL clauses in alphabetical order, then SQL operators in alphabetical order, and finally, SQL functions in alphabetical order.
    You need to end each of your SQL queries with a semicolon.
- **Note:** Anything you can write using relational algebra you can write in SQL, but not everything you can write in SQL can be written in relational algebra.

SQL Statements in Alphabetical Order:
Alter Table:
- The ALTER TABLE statement is used to add, delete (drop), or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.
- To add a column in a table, use the following syntax:
    ALTER TABLE table_name ADD column_name datatype;
- To delete a column in a table, use the following syntax:
    ALTER TABLE table_name DROP COLUMN column_name;
- To change the data type of a column in a table, use the following syntax:
    ALTER TABLE table_name ALTER COLUMN column_name datatype;

AS:

- The as statement creates an alias for a table or a column.
  SQL aliases are used to give a table, or a column in a table, a temporary name.
- Aliases are often used to make column names more readable.
- An alias only exists for the duration of the query.
- **Syntax for column: SELECT column_name AS alias_name FROM table_name;**
- E.g.
  Consider the table below:

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

  If I run the query **select FirstName as FName from Students;**, I get

	FName
1	Rick
2	ABC
3	Rick
4	Rick
5	ABC

- **Syntax for table: SELECT column_name(s) FROM table_name AS alias_name;**
- **Note:** There's another way we can rename tables, shown below.
  E.g. **SELECT e.name, d.name FROM employee e, department d WHERE d.name = 'marketing' AND e.name = 'Horton';**

Create Table:

- The create table statement creates a table with the specified table name and column name(s).
- **Syntax: CREATE TABLE table_name(column1 datatype, column2 datatype, column3 datatype, ..., column(n) datatype);**
- E.g.
 The query **create table Students(FirstName Text, LastName Text, StudentNumber INTEGER);**
  Creates this table

Table:	Students		∨
FirstName	LastName	StudentNumber	
Filter	Filter	Filter	

- The column parameters specify the names of the columns of the table.
- The datatype parameter specifies the type of data the column can hold (e.g. text, integer, date, etc).

Cross Join:
- Same as cartesian join.
- **Syntax: SELECT columns FROM Table1 CROSS JOIN Table2;**
- **Note:** You can do **SELECT columns FROM Table1,Table2;** to get the same result. When you have a comma between tables names, you are getting the cartesian join of the tables.
 I.e.
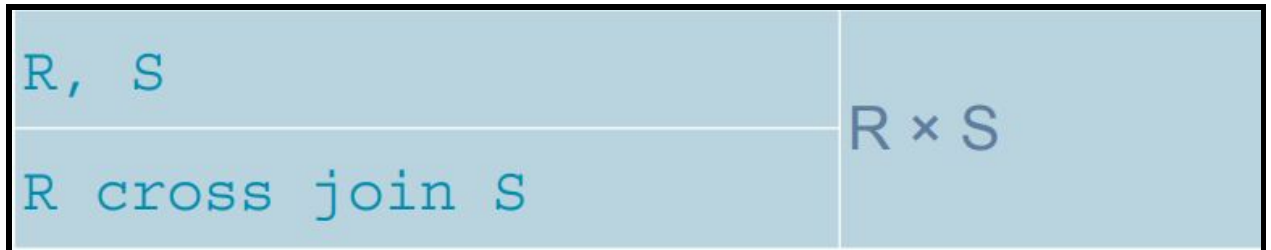
Delete:
- The DELETE statement is used to delete existing records in a table.
- **Syntax: DELETE FROM table_name WHERE condition;**
- **Note:** Be careful when deleting records in a table. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted.

Drop Table:
- The DROP TABLE statement is used to drop (delete) an existing table in a database.
- **Syntax: DROP TABLE table_name;**

Full Join:
- Also known as FULL OUTER JOIN.
- The FULL JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.
  I.e.
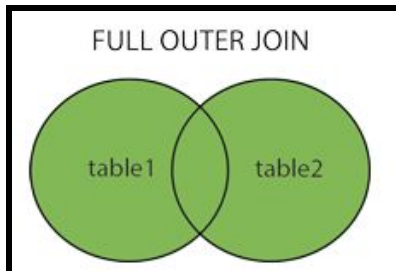
- **Note:** FULL OUTER JOIN can potentially return very large result-sets.
- **Note:** FULL OUTER JOIN and FULL JOIN are the same.
- **Syntax: SELECT column_name(s) FROM table1 FULL OUTER JOIN table2 ON table1.column_name = table2.column_name WHERE condition;**

Group By:
- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- **Syntax: SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s)**
- E.g. Consider the table below

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select AVG(Mark) from Marks group by StudentNumber;**, I get

	AVG(Mark)
1	82.3333333333333
2	75.5
3	71.5
4	82.0
5	52.0

Inner Join:
- The INNER JOIN keyword selects records that have matching values in both tables. I.e.

INNER JOIN
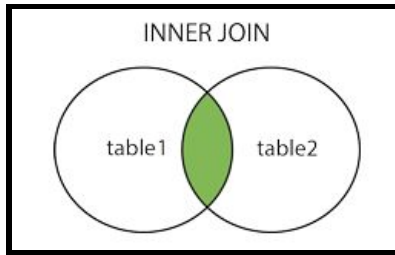
table1   table2

- **Syntax: SELECT column_name(s) FROM table1 INNER JOIN table2 ON table1.column_name = table2.column_name;**
- E.g.
Consider the tables

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select * from Students inner join Marks on Students.StudentNumber = Marks.StudentNumber;**, I get

	FirstName	LastName	StudentNumber	StudentNumber	Mark
1	Rick	Lan	99	99	95
2	ABC	DEF	100	100	74
3	Rick	DEF	101	101	88
4	Rick	XYZ	102	102	82
5	ABC	XYZ	103	103	52

Insert Into:
- The INSERT INTO statement is used to insert new records in a table.
- **Syntax #1: INSERT INTO table_name (column1, column2, column3, …, column(n)) VALUES (value1, value2, value3, …, value(n));**
- **Syntax #2: INSERT INTO table_name VALUES (value1, value2, value3, …, value(n));**
- **Syntax #3: INSERT INTO table_name (subquery);**
- The first way specifies both the column names and the values to be inserted.
- You use the second way if you are adding values for all the columns of the table. Here, you do not need to specify the column names. However, make sure the order of the values is in the same order as the columns in the table.
- Sometimes we want to insert tuples, but we don't have values for all attributes. If we name the attributes we are providing values for, the system will use NULL or a default for the rest.
- E.g. Currently, the Students table is empty.

Table: Students

FirstName	LastName	StudentNumber
Filter	Filter	Filter

However, if I run the query **insert into Students values ("Rick", "Lan", 100);**, then the table becomes

Table: Students

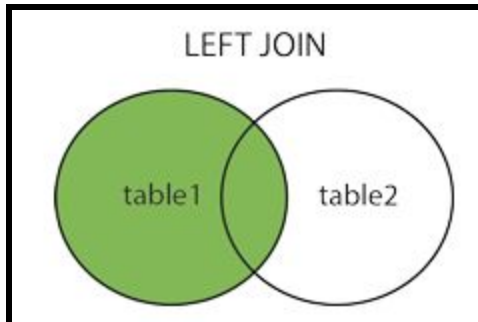|	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100

- E.g. If I run the query **insert into Students(FirstName, LastName) values ("ABC", "DEF");**, the table becomes

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	NULL

Left Join:
- Also known as LEFT OUTER JOIN.
- The LEFT JOIN statement returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.
  I.e.

LEFT JOIN

table1   table2

- **Syntax: SELECT column_name(s) FROM table1 LEFT JOIN table2 ON table1.column_name = table2.column_name;**
- E.g.
  Consider the tables below:

Table: Marks

StudentNumber	Mark
Filter	Filter
1 99	95
2 100	74
3 101	88
4 102	82
5 104	52

Table: Students

FirstName	LastName	StudentNumber
Filter	Filter	Filter
1 Rick	Lan	99
2 ABC	DEF	100
3 Rick	DEF	101
4 Rick	XYZ	102
5 ABC	XYZ	103

If I do the query **select * from Students left join Marks on Students.StudentNumber = Marks.StudentNumber;**, I get

	FirstName	LastName	StudentNumber	StudentNumber	Mark
1	Rick	Lan	99	99	95
2	ABC	DEF	100	100	74
3	Rick	DEF	101	101	88
4	Rick	XYZ	102	102	82
5	ABC	XYZ	103	NULL	NULL

Natural Join:
- **Syntax: SELECT columns FROM Table1 NATURAL JOIN Table2;**
  I.e.

R natural join S	R ⋈ S

- E.g. Consider the tables below

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **SELECT * FROM Students NATURAL JOIN Marks;**, I get

	FirstName	LastName	StudentNumber	Mark
1	Rick	Lan	99	65
2	Rick	Lan	99	87
3	Rick	Lan	99	95
4	ABC	DEF	100	74
5	ABC	DEF	100	77
6	Rick	DEF	101	55
7	Rick	DEF	101	88
8	Rick	XYZ	102	82

- In practice, natural joins are brittle. A working query can be broken by adding a column to a schema. Furthermore, having implicit comparisons impairs readability. The best practise is not to use natural joins.
- **Note:** We can also do **SELECT columns FROM Table1 NATURAL LEFT|RIGHT FULL JOIN Table2;**
  E.g. If I do the query **select * from Students natural left join Marks;**, I get

	FirstName	LastName	StudentNumber	Mark
1	Rick	Lan	99	65
2	Rick	Lan	99	87
3	Rick	Lan	99	95
4	ABC	DEF	100	74
5	ABC	DEF	100	77
6	Rick	DEF	101	55
7	Rick	DEF	101	88
8	Rick	XYZ	102	82
9	ABC	XYZ	103	NULL

Order By:
- The ORDER BY statement is used to sort the result-set in ascending or descending order.
- The ORDER BY statement sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.
- The ordering is the last thing done before the SELECT, so all attributes are still available.
- **Syntax: SELECT column1, column2, …, column(n) FROM table_name ORDER BY column1, column2, ... ASC|DESC;**
- The attribute list can include expressions: e.g. **ORDER BY sales+rentals**.
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from students order by FirstName;**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	99
2	ABC	XYZ	103
3	Rick	Lan	100
4	Rick	DEF	101
5	Rick	XYZ	102

If I run the query **select * from students order by FirstName, LastName;**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	99
2	ABC	XYZ	103
3	Rick	DEF	101
4	Rick	Lan	100
5	Rick	XYZ	102

Note: Because I didn't specify ascending or descending in the query above, the default is ascending.

If I run the query **select * from students order by FirstName DESC;**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	100
2	Rick	DEF	101
3	Rick	XYZ	102
4	ABC	DEF	99
5	ABC	XYZ	103

Note: If you order by more than 1 column, you can order some columns in ascending order and others in descending order.

If I run the query **select * from students order by FirstName DESC, LastName ASC;**, I get

	FirstName	LastName	StudentNumber
1	Rick	DEF	101
2	Rick	Lan	100
3	Rick	XYZ	102
4	ABC	DEF	99
5	ABC	XYZ	103

Right Join:
- Also known as RIGHT OUTER JOIN.
- The RIGHT JOIN statement returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.
  I.e.
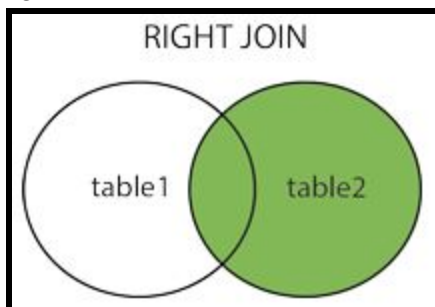
- **Syntax: SELECT column_name(s) FROM table1 RIGHT JOIN table2 ON table1.column_name = table2.column_name;**

Select:
- The select statement gets the specified columns from a table.
- **Syntax: select (column names) from table**
- **Note:** If you want to get all the columns from a table, you can do
  select * from table(s)
  A * in the SELECT clause means all attributes of this relation.
- E.g. Currently, the Students table looks like this

If I run the query **select * from Students;**, I get

If I run the query **select FirstName from Students;**, I get

- **Note:** If you have multiple tables after the from keyword, you will get a cartesian product of those tables.
  E.g. The query **SELECT cNum FROM Offering, Took WHERE Offering.id = Took.oid and dept = 'CSC';** is analogous to $\pi_{cNum}(\sigma_{Offering.id=Took.id \wedge dept='csc'}(Offering \times Took))$.
  In SQL, Offering, Took is the same as Offering x Took.
- **Note:** Instead of a simple attribute name, you can use an expression in a SELECT clause.
  E.g.
  SELECT sid, grade+10 AS adjusted FROM Took;
  SELECT dept||cnum FROM course; Note: || means concatenate.

Select Distinct:
- The SELECT DISTINCT statement is used to return only distinct (different) values.
- Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.
- **Syntax: SELECT DISTINCT column1, column2, …, column(n) FROM table_name;**

Self Join:
- A self JOIN is a regular join, but the table is joined with itself.
- **Syntax: SELECT column_name(s) FROM table1 T1, table1 T2 WHERE condition;**
- **Note:** T1 and T2 are different table aliases for the same table.
- E.g. **SELECT e1.name, e2.name FROM employee e1, employee e2 WHERE e1.salary < e2.salary;**

Theta Join:
- **Syntax: SELECT columns FROM Table1 JOIN Table2 ON condition;**
  I.e.

R join S on Condition $R \bowtie_{condition} S$

- E.g. Consider the tables below

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **SELECT FirstName, LastName, avg(Mark) FROM Students JOIN Marks on Students.StudentNumber = Marks.StudentNumber GROUP BY Marks.StudentNumber HAVING avg(Mark) > 80;**, I get:

	FirstName	LastName	avg(Mark)
1	Rick	Lan	82.3333333333333
2	Rick	XYZ	82.0

Update:
- The UPDATE statement is used to modify the existing records in a table.
- **Syntax: UPDATE table_name SET column1 = value1, column2 = value2, …, column(n) = value(n) where condition;**
- **Note:** Be careful when updating records in a table. The WHERE clause, while optional, specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated.
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **update Students set StudentNumber = 99 where FirstName = "Rick" and LastName = "Lan";**, the table becomes

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	99
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **update Students set StudentNumber = 100 where FirstName = "ABC" and LastName = "DEF";**, the table becomes

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

View:
- A **view** is a relation defined in terms of stored tables called **base tables** and other views.
- We can access a view like any table.
- There are 2 kinds of view:
    1. **Virtual:** No tuples are stored. The view is just a query for constructing the relation when needed.
    2. **Materialized:** The table is actually constructed and stored. It is expensive to maintain.
- A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.
- Views, which are a type of virtual tables allow users to do the following:
    - Break down a large query.
    - Provide another way of looking at the same data, e.g. for one category of user.
    - Structure data in a way that users or classes of users find natural or intuitive.
    - Restrict access to the data in such a way that a user can see and sometimes modify exactly what they need and no more.
    - Summarize data from various tables which can be used to generate reports.
- **Syntax: CREATE VIEW view_name AS (SELECT QUERY);**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **CREATE VIEW StudentNames AS SELECT FirstName, LastName FROM Students;**, I get

Table: StudentNames

	FirstName	LastName
	Filter	Filter
1	Rick	Lan
2	ABC	DEF
3	Rick	DEF
4	Rick	XYZ
5	ABC	XYZ

- Generally, it is impossible to modify a virtual view, because it doesn't exist. Furthermore, most systems prohibit most view updates.
- A problem is that each time a base table changes, the materialized view may change and we can not afford to recompute the view with each change. A solution is to do periodic reconstructions of the materialized view, which is otherwise out of date.

SQL Clauses in Alphabetical Order:
Having:
- The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions (avg, count, max, min, sum, etc).
- **Syntax: SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s) HAVING condition**
- **Note:** The having clause is always used with the group by statement.
- E.g. Consider the table below:

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select StudentNumber from Marks group by StudentNumber having AVG(Mark) > 70;**, I get

	StudentNumber
1	99
2	100
3	101
4	102

- Outside subqueries, HAVING may refer to attributes only if they are either:
  - aggregated or
  - an attribute on the GROUP BY list.

Limit:
- The LIMIT clause is used to specify the number of records to return.
- E.g. With Select
  If I run the query **select * from Students limit 1;**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99

Where:
- The WHERE clause is used to extract only those records that fulfill a specified condition. We can build boolean expressions with operators that produce boolean results. Some comparison operators are:
    - <> (Means not equal to)
    - =
    - <
    - >
    - <=
    - >=
- It can be used with select, update, delete, and other statements.
- E.g. With Select
  Consider the table below:

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	100
2	ABC	DEF	NULL
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

  If I run the query **select FirstName from Students where FirstName = "Rick";**, I get

	FirstName
1	Rick
2	Rick
3	Rick

- If I run the query **select StudentNumber from Students where FirstName = "Rick";**, I get

	StudentNumber
1	100
2	101
3	102

- The WHERE clause can be combined with **AND**, **OR**, and **NOT** operators.
- The AND operator displays a record if all the conditions separated by AND are true.
- The OR operator displays a record if any of the conditions separated by OR is true.
- The NOT operator displays a record if the condition(s) is NOT true.

- E.g.
  Consider the table below:

If I run the query **select LastName from Students where NOT FirstName = "Rick";**, I get

If I run the query **update Students set StudentNumber = 99 where FirstName="ABC" and LastName="DEF";**, the table becomes

- The WHERE clause can also be used with the **LIKE** operator to search for a specified pattern in a column.
- There are two wildcards often used in conjunction with the LIKE operator:
    1. %: The percent sign represents zero, one, or multiple characters.
    2. _: The underscore represents a single character.
- **Syntax: SELECT column1, column2, …, column(n) FROM table_name WHERE column(N) LIKE pattern;**

Detailed

- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where FirstName like "A%";**, I get

	FirstName	LastName	StudentNumber
1	ABC	DEF	100
2	ABC	XYZ	103

If I run the query **select * from Students where FirstName like "R_CK";**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	Rick	DEF	101
3	Rick	XYZ	102

- The WHERE clause can be used with the **IN** operator to specify multiple values.
  The IN operator is a shorthand for multiple OR conditions.
- **Syntax #1: SELECT column_name(s) FROM table_name WHERE column_name IN (value1, value2, ...);**
- **Syntax #2: SELECT column_name(s) FROM table_name WHERE column_name IN (SELECT STATEMENT);**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where StudentNumber in (99, 100, 101);**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101

- The WHERE clause can be used with the BETWEEN operator to select values within a given range. The values can be numbers, text, or dates.
  The BETWEEN operator is inclusive: begin and end values are included.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;**
- E.g. Consider the table

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

If I run the query **select * from Students where StudentNumber between 99 and 101;**, I get

	FirstName	LastName	StudentNumber
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101

SQL Operators in Alphabetical Order:

All:
- The ALL operator is used with a WHERE or HAVING clause.
- The ALL operator returns true if all of the subquery values meet the condition.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name operator ALL (SELECT column_name FROM table_name WHERE condition);**
- **Note:** The operator must be a standard comparison operator (=, <>, >, >=, <, or <=).

Any:
- The ANY operator is used with a WHERE or HAVING clause.
- The ANY operator returns true if any of the subquery values meet the condition.
- **Syntax: SELECT column_name(s) FROM table_name WHERE column_name operator ANY (SELECT column_name FROM table_name WHERE condition);**
- **Note:** The operator must be a standard comparison operator (=, <>, >, >=, <, or <=).
- The ANY operator is also called SOME.
  I.e. ANY is equivalent to SOME.

Except:
- The SQL EXCEPT operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in the second SELECT statement.
- Each SELECT statement within EXCEPT must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: (select column1, column2, …, column(n) from table1) except (select column1, column2, …, column(n) from table1);**
- E.g. Consider the tables below

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

StudentNumber	Mark
Filter	Filter
1 99	95
2 99	87
3 99	65
4 100	74
5 100	77
6 101	88
7 101	55
8 102	82
9 104	52

If I run the query **select StudentNumber from Students EXCEPT select StudentNumber from Marks;**, I get

StudentNumber
1 103

Exists:
- The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns true if the subquery returns one or more records. The subquery is a SELECT statement. If the subquery returns at least one record in its result set, the EXISTS clause will evaluate to true and the EXISTS condition will be met. If the subquery does not return any records, the EXISTS clause will evaluate to false and the EXISTS condition will not be met.
- **Syntax: SELECT column_name(s) FROM table_name WHERE EXISTS (SELECT column_name FROM table_name WHERE condition);**

Intersect:
- The SQL INTERSECT operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
- Each SELECT statement within INTERSECT must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: SELECT column_name(s) FROM table1 intersect SELECT column_name(s) FROM table2;**
- E.g.
  Consider the tables below:

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52

If I run the query **select StudentNumber from Students intersect SELECT StudentNumber from Marks;**, I get

	StudentNumber
1	99
2	100
3	101
4	102

Union:
- The UNION operator is used to combine the result-set of two or more SELECT statements.
- Each SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types.
- The columns in each SELECT statement must also be in the same order.
- **Syntax: SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;**
- E.g.
  Consider the tables below:

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	99
2	ABC	DEF	100
3	Rick	DEF	101
4	Rick	XYZ	102
5	ABC	XYZ	103

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52

If I run the query **select StudentNumber from Students union SELECT StudentNumber from Marks;**, I get

	StudentNumber
1	99
2	100
3	101
4	102
5	103
6	104

- The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.
  Syntax: SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;
- E.g. Using the 2 tables from above, if I run the query **select StudentNumber from Students union all SELECT StudentNumber from Marks;**, I get

	StudentNumber
1	99
2	100
3	101
4	102
5	103
6	99
7	100
8	101
9	102
10	104

SQL Functions in Alphabetical Order:
Note: These functions are called **aggregate functions**.
If any aggregation is used, then each element of the SELECT list must be either be:
1. aggregated or
2. an attribute on the GROUP BY list.

Otherwise, it doesn't even make sense to include the attribute.
AVG:
- The AVG() function returns the average value of a numeric column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select avg(mark) from Marks;**, I get

	avg(mark)
1	78.2

COUNT:
- The COUNT() function returns the number of rows that matches a specified criterion.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select count(mark) from Marks;**, I get

	count(mark)
1	5

- You can do **COUNT(*)** to get the number of tuples.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select count(*) from Marks;**, I get

	count(*)
1	9

MAX:
- The MAX() function returns the largest value of the selected column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query **select max(mark) from Marks;**, I get

	max(mark)
1	95

MIN:
- The MIN() function returns the smallest value of the selected column.
- E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I run the query select min(mark) from Marks;, I get

	min(mark)
1	52

SUM:
-   The SUM() function returns the total sum of a numeric column.
-   E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	103	52

If I do the query **select sum(mark) from Marks;**, I get

	sum(mark)
1	391

Note: You can use the keyword **DISTINCT** with any of these functions to prevent counting duplicates.
E.g. **COUNT (DISTINCT column)** or **SUM(DISTINCT column)**, etc.
DISTINCT does not affect MIN or MAX.
E.g. Consider the table

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	99	87
3	99	65
4	100	74
5	100	77
6	101	88
7	101	55
8	102	82
9	104	52

If I run the query **select count(StudentNumber) from Marks;**, I get

	count(StudentNumber)
1	9

But, if I run the query **select count(DISTINCT StudentNumber) from Marks;**, I get

	count(DISTINCT StudentNumber)
1	5

Order of execution of a SQL query:

Query order	Execution order
SELECT	FROM
FROM	WHERE
WHERE	GROUP BY
GROUP BY	HAVING
HAVING	SELECT
ORDER BY	ORDER BY

Set Operations:
- Tables can have duplicates in SQL.
- A table can have duplicate tuples, unless this would violate an integrity constraint.
- SELECT-FROM-WHERE statements leave duplicates in unless you say not to. This is because:
    - Getting rid of duplicates is expensive.
    - We may want the duplicates because they tell us how many times something occurred.
- SQL treats tables as **bags/multisets** rather than sets.
- Bags are just like sets, but duplicates are allowed.
  E.g.
  {6, 2, 7, 1, 9} is a set (and a bag) while {6, 2, 2, 7, 1, 9, 9} is not a set but is a bag.
- Like with sets, order doesn't matter with bags.
- **Note:** In SQL, Union, Intersect and Except use set semantics by default. This means that duplicates are eliminated from the result.
- Consider Union, denoted as U for this example, Intersect, denoted as ∩ for this example, and Except, denoted as - for this example and suppose they use bag semantics instead of set semantics. Furthermore, suppose tuple t occurs m times in relation R and n times in relation S.
  Then, we have:

Operation	Number of occurrences of t in result
R ∩ S	$\min(m, n)$
R ∪ S	$m + n$
R - S	$\max(m-n, 0)$

E.g. of union, intersect and except using bag semantics:
1. {1, 1, 1, 3, 7, 7, 8} ∪ {1, 5, 7, 7, 8, 8} = {1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8}
2. {1, 1, 1, 3, 7, 7, 8} ∩ {1, 5, 7, 7, 8, 8} = {1, 7, 7, 8}
3. {1, 1, 1, 3, 7, 7, 8} − {1, 5, 7, 7, 8, 8} = {1, 1, 3}

- **Note:** We can force the result of a Select-From-Where query to be a set by using **SELECT DISTINCT**.
- **Note:** We can force the result of a set operation to be a bag by using the keyword **ALL**.

Dangling tuples:
- With joins that require some attributes to match, tuples lacking a match are left out of the results. We say that they are **dangling**.
- An **outer join** preserves dangling tuples by padding them with NULL in the other relation.
  There are 3 types of outer joins.
    1. LEFT OUTER JOIN
    2. RIGHT OUTER JOIN
    3. FULL OUTER JOIN
-
- A join that doesn't pad with NULL is called an **inner join**.
- There are keywords INNER and OUTER, but you never need to use them.
- You get an outer join iff you use the keywords LEFT, RIGHT, or FULL.
- If you don't use the keywords LEFT, RIGHT, or FULL you get an inner join.
- Here is a chart to show comparisons:

	Theta Join	**Natural Join**
Inner Join	A JOIN B ON C	A NATURAL JOIN B
Outer Join	A {LEFT|RIGHT|FULL} JOIN B ON C	A NATURAL {LEFT|RIGHT|FULL} JOIN B

- E.g. Consider the tables R and S below:

R	A	B
	1	2
	4	5

S	B	C
	2	3
	6	7

This is **R NATURAL JOIN S**:

A	B	C
1	2	3

This is **R NATURAL FULL JOIN S**:

A	B	C
1	2	3
4	5	NULL
NULL	6	7

This is **R NATURAL LEFT JOIN S**:

A	B	C
1	2	3
4	5	NULL

This is **R NATURAL RIGHT JOIN S**:

A	B	C
1	2	3
NULL	6	7

Null Values:
- There are 2 common scenarios for null values:
    1. Missing value
       E.g. We know a student has some email address, but we don't know what it is.
    2. Inapplicable attribute
       E.g. The value of the attribute spouse is inapplicable for an unmarried person.
- One possibility for representing missing information is to use a special value as a placeholder. However, a better solution is to use a value not in any domain. We call this a **null value**.

- Tuples in SQL relations can have NULL as a value for one or more components.
- You can compare an attribute value to NULL with the following clauses:
    1. **IS NULL**
    2. **IS NOT NULL**
    E.g. **SELECT * FROM Course WHERE breadth IS NULL;**
- Because of NULL, we need three truth-values:
    1. If one or both operands to a comparison is NULL, the comparison always evaluates to UNKNOWN.
    2. True
    3. False
- We need to know how the three truth-values combine with AND, OR and NOT.
- To do so, we can think in terms of numbers:
    - TRUE = 1
    - FALSE = 0
    - UNKNOWN = 0.5
- AND is min, OR is max, NOT x is (1-x).
- **Note:** A tuple is in a query result iff the WHERE clause is TRUE. UNKNOWN is not good enough.
    E.g. Consider the table below:

	StudentNumber	Mark
	Filter	Filter
1	99	95
2	100	74
3	101	88
4	102	82
5	104	52
6	99	87
7	100	77
8	99	65
9	101	55
10	101	NULL

If I run the query **select * from Marks where Mark > 70;**, I get

	StudentNumber	Mark
1	99	95
2	100	74
3	101	88
4	102	82
5	99	87
6	100	77

- **Note:** The aggregate functions ignores NULL.
- NULL never contributes to a sum, average, or count, and NULL can never be the minimum or maximum of a column unless every value is NULL.
- If there are no non-NULL values in a column, then the result of the aggregation is NULL. An exception is that COUNT of an empty set is 0.
- E.g. Consider the table below

Table:	dummy
number
Filter
1	NULL
2	0

If I run the query **select count(number) from dummy;**, I get

count(number)
1	1

If I run the query **select min(number) from dummy;**, I get

min(number)
1	0

However, I change the table such that all the rows are null,

Table:	dummy
number
Filter
1	NULL

and I run the query **select count(number) from dummy;**, I get

count(number)
1	0

If I run the query **select min(number) from dummy;**, I get

min(number)
1	NULL

If I run the query **select count(*) from dummy;**, I get

count(*)
1	1

- Here's a table of the summary of aggregate functions on NULL

	some nulls in A	All nulls in A
min(A)		
max(A)		null
sum(A)	ignore the nulls	
avg(A)		
count(A)		0
count(*)	all tuples count	

- Other corner cases to think about:
    1. SELECT DISTINCT: Are 2 NULL values equal?
       (For the most part, for select distinct, 2 null values are equal.)
    2. NATURAL JOIN: Are 2 NULL values equal?
       (For the most part, for natural join, 2 null values are not equal.)
    3. SET OPERATIONS: Are 2 NULL values equal?
       (For the most part, for set operations, 2 null values are equal.)
    4. UNIQUE Constraint: Do 2 NULL values violate it?

    However, this behaviour may vary across DBMSs.

    Different DBMSs have different implementations.

Subqueries:
Can go:
- In a FROM clause:
    - In place of a relation name in the FROM clause, we can use a subquery.
    - The subquery must be parenthesized.
    - We must name the result, so you can refer to it in the outer query.
    - E.g.
      SELECT sid, dept||cnum as course, grade FROM Took,
      (SELECT * FROM Offering WHERE instructor='Horton') Hoffering
      WHERE Took.oid = Hoffering.oid;
- In a WHERE clause:
    - If a subquery is guaranteed to produce exactly one tuple, then the subquery can be used as a value.
    - The simplest situation is that one tuple has only one component.
    - E.g.
      SELECT sid, surname FROM Student WHERE cgpa >
      (SELECT cgpa FROM Student WHERE sid = 99999);
    - When a subquery can return multiple values, we can make comparisons using a quantifier by using the ALL, ANY/SOME, IN, and EXISTS operators.
- As operands to UNION, INTERSECT or EXCEPT.

Scope:
- Queries are evaluated from the inside out.
- If a name might refer to more than one thing, use the most closely nested one.
- If a subquery refers only to names defined inside it, it can be evaluated once and used repeatedly in the outer query.

- If it refers to any name defined outside of itself, it must be evaluated once for each tuple in the outer query. These are called **correlated subqueries**.
  Think of this as a nested loop.
- Renaming can make scope explicit.
  E.g.
  SELECT instructor FROM Offering Off1 WHERE NOT EXISTS
  (SELECT * FROM Offering Off2 WHERE Off2.oid <> Off1.oid AND Off2.instructor =
  Off1.instructor);