SQL

What is SQL?

SQL

Structured Query Language

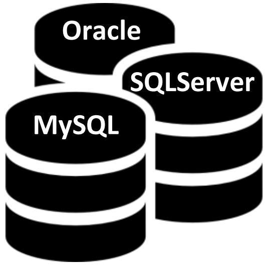
A querying language designed for accessing and manipulating information from Relational Databases

SQL

Advantages

Platform Independent

Portable



Database Independent

Simple Syntax

Quick and Efficient Retrieval

SQL

Advantages

- SQL is portable and can be used independent of the platform
- Can be used for querying data in a wide variety of databases and data repositories,
- Has a simple syntax that is similar to the english language
- Can retrieve large amounts of data quickly and efficiently
- Runs on an interpreter system (which means code can be executed as soon as it is written, making prototyping quick and easy)

SQL

Command Types

DQL
(Data
Query
Language)

SELECT

DDL
(Data
Definition
Language)

CREATE
DROP
ALTER
RENAME

DML
(Data
Manipulation
Language)

INSERT
UPDATE
DELETE

TCL
(Transaction
Control
Language)

COMMIT
ROLLBACK
SAVEPOINT

DCL
(Data
Control
Language)

GRANT
REVOKE

SQL

Command Types

- DDL: Data Definition Language
  - Define admissible database content (schema)
- DQL: Data Query Language
  - Query and retrieve database content
- DML: Data Manipulation Language
  - Change and retrieve database content
- TCL: Transaction Control Language
  - Groups SQL commands (transactions)
- DCL: Data Control Language
  - Assign data access rights

DQL

SQL

DQL

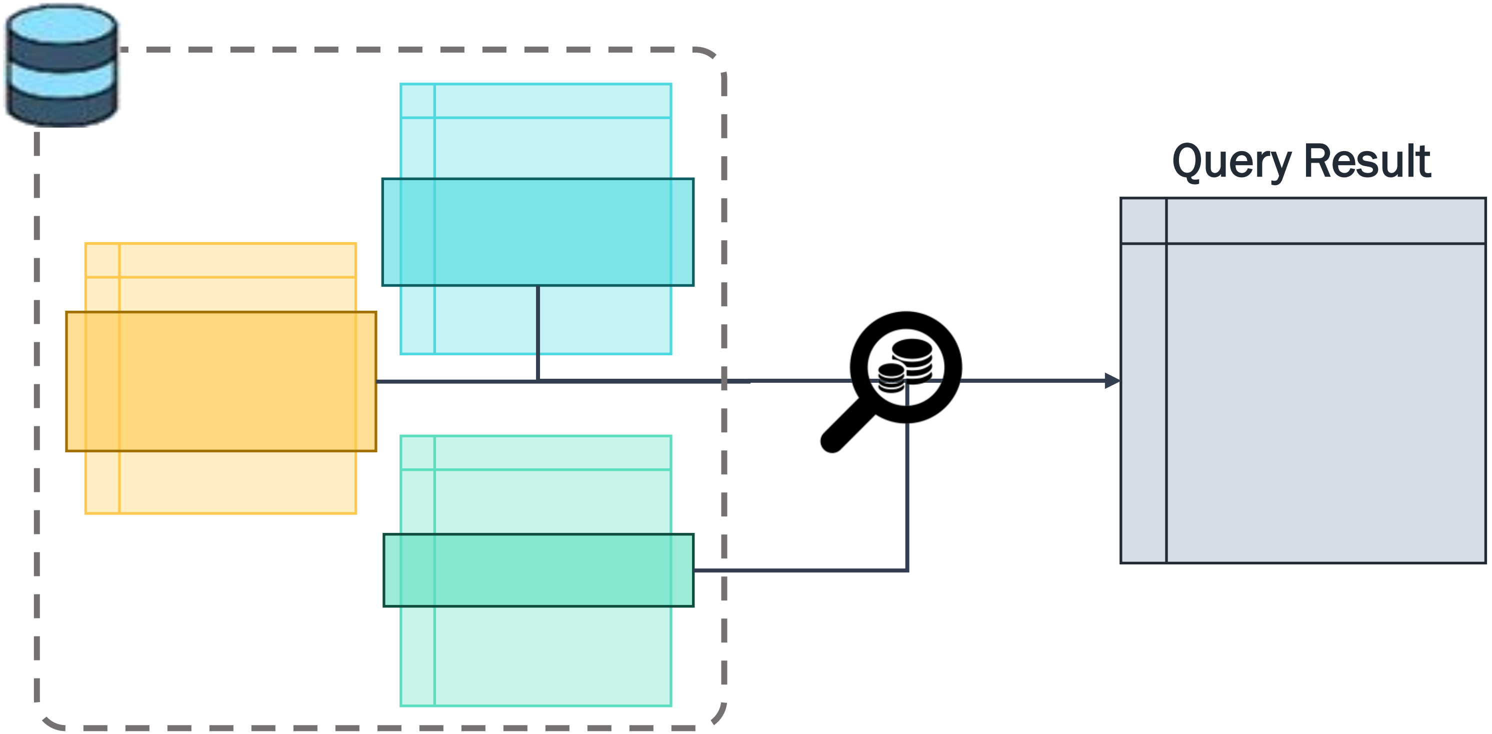
DQL
Data Query Language

Query and retrieve data from database

Describes a new relation to generate

DQL
(Data
Query
Language)

SELECT



SQL

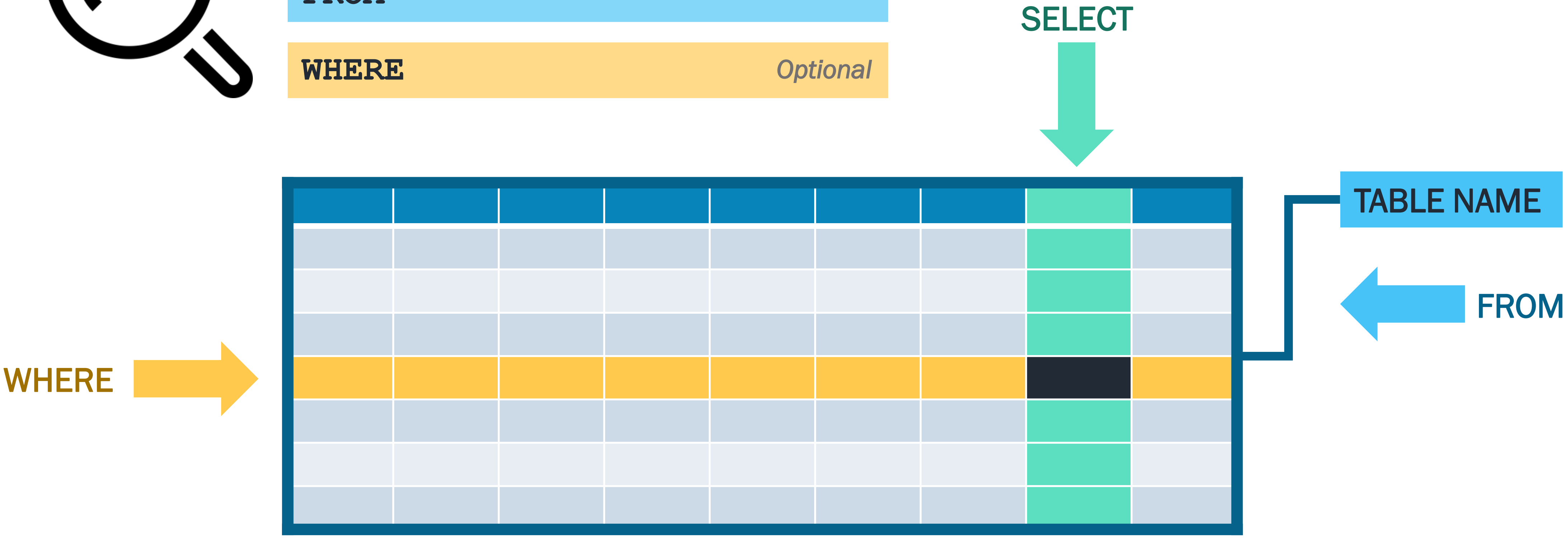
DQL

A simple SQL Query consists of 3 main clauses:

SELECT

FROM

WHERE *Optional*



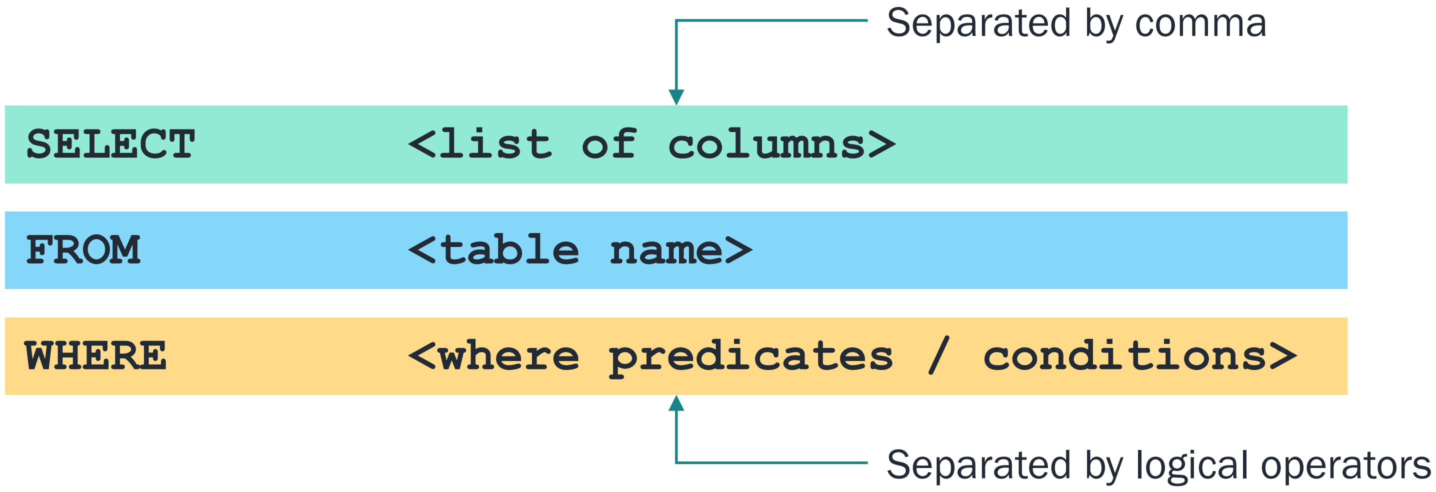
SQL

DQL

- What is an SQL query? A SQL query describes a new relation to generate
- A simple SQL query consist of 3 clauses:
  - SELECT: describes the columns of relation to generate
  - FROM: describes source relations and how to match
  - WHERE: defines conditions result rows must satisfy
- The syntax is:
  - SELECT <column_list>
  - FROM <table1_name> JOIN <table2_name> ON (<join_predicate>)
  - WHERE <where_predicate>
- where <column_list> is a comma separated list of columns and <table1_name> and <table2_name> are database relations, <join_predicate> is a condition defining matching tuples pairs and <where_predicate> are additional conditions.

SQL

DQL



Separated by comma

```
SELECT        <list of columns>
FROM          <table name>
WHERE         <where predicates / conditions>
```

Separated by logical operators

Example:

```
SELECT        first_name, last_name
FROM          student_table
WHERE         student_id = 13787545
```

SQL

DQL : *

```
SELECT          *
```

Selects (retrieves) all the columns from a table

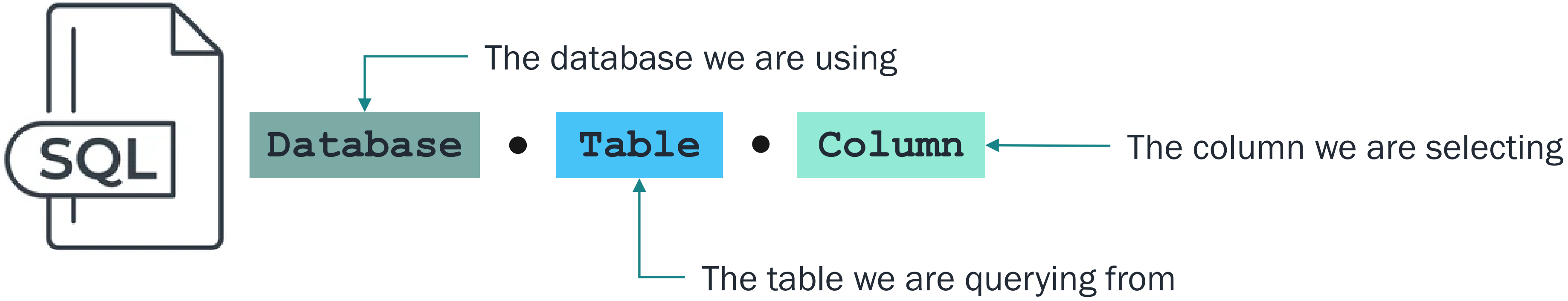
Example:

```
SELECT          *
FROM            customers
```

SQL

DQL : .

A dot (.) can be used to specify objects in hierarchical form



Database • Table • Column

The database we are using

The table we are querying from

The column we are selecting

Example:

```
SELECT        coffeeshop.customers.customer_id
FROM          coffeeshop.customers
```

SQL

Aliases

Aliases are used to give a column a temporary name while in a query

```
SELECT          column1 AS c1
```

Example:

```
SELECT          loyalty_card_number AS lcn
FROM            customers
```

SQL

Aliases

Aliases are also used to give a table a temporary name while in a query

```
FROM            table1 AS t1
```

Example:

```
SELECT          *
FROM            customers AS c
```

```
SELECT          c.customer_id
FROM            customers AS c
```

SQL

DQL: Distinct

Used to return only distinct (different) values (to eliminate duplicates)

```
SELECT       DISTINCT Column1
```

Example:

```
SELECT       DISTINCT continent
FROM         country
```

```
SELECT       DISTINCT GovernmentForm
FROM         country
```

SQL

DQL: Order By

Used to sort the outcome table by a set of columns

```
SELECT        <list of columns>
FROM          <table name>
ORDER BY      <list of columns>
```

By default, is in Ascending order
Use **DESC** to sort Descendingly

Example:

```
SELECT        *
FROM          country
ORDER BY      population DESC
```

SQL

DQL: Limit

Used to limit the number of rows returned

```
SELECT        <list of columns>
FROM          <table name>
LIMIT         <number>
```

Example:

```
SELECT        *
FROM          country
LIMIT         10
```

Operators

SQL

Operators

Arithmetic Operators

+	-	*	/	%

Addition, Subtraction, Multiplication, Division, Modulus

Example:

```
SELECT        current_wholesale_price,
              3*(current_wholesale_price+2) AS new_price
FROM          coffeeshop.products;
```

```
SELECT        *
FROM          products
WHERE         current_wholesale_price + 3 < 10
```

SQL

Operators

Comparison Operators

= > < >= <= <> !

Used to compare the values of two operands

Example:

```
SELECT        *
FROM          products
WHERE         tax_exempt_yn = ‘Y’
```

```
SELECT        *
FROM          products
WHERE         current_wholesale_price + 3 < 10
```

SQL

Operators

Logical Operators

AND

TRUE if all the conditions separated by AND is TRUE

Condition 1	AND	Condition 2	›	Result
TRUE		TRUE		TRUE
TRUE		FALSE		FALSE
FALSE		FALSE		FALSE

SQL

Operators

Logical Operators

OR

TRUE if any of the conditions separated by OR is TRUE

Condition 1	OR	Condition 2	Result
TRUE		TRUE	TRUE
TRUE		FALSE	TRUE
FALSE		FALSE	FALSE

SQL

Operators

Logical Operators

NOT

Used to negate (reverse) the meaning of a comparison or a logical operator

Examples:

NOT	a > b	≡	a <= b	or	! a > b
NOT	a = b	≡	a != b	or	a <> b

NOT	Condition	›	Result
NOT	TRUE		FALSE
NOT	FALSE		TRUE

SQL

Unknown Values

NULL Values

NULL

Represents missing, unknown, or blank

Arithmetic with NULL values:

NULL + - * / % anything = NULL

Comparison with NULL values:

NULL = > < >= <= <> anything ≡ NULL

SQL

Operators

Logical Operators

IS   IS NOT

Used to check if a value is NULL or not

Examples:

anything IS NULL ≡ TRUE or FALSE

anything IS NOT NULL ≡ TRUE or FALSE

Exercise: Can these expressions ever result in NULL? Why?

SQL

Three-Valued Logic

Ternary Outcomes

TRUE FALSE NULL

Outcome of an expression can be evaluated as either TRUE, FALSE, or NULL

Condition 1	OR	Condition 2
TRUE		NULL
FALSE		NULL

Condition 1	AND	Condition 2
TRUE		NULL
FALSE		NULL

Exercise: What are outcomes of these expressions?

SQL

Operators - Exercise

What are the outcomes of the following expressions?

1	`SELECT`	`4 = NULL`
2	`SELECT`	`NULL = NULL`
3	`SELECT`	`NULL IS NULL`
4	`SELECT`	`NULL IS NOT NULL`
5	`SELECT`	`TRUE OR NULL`
6	`SELECT`	`FALSE OR NULL`
7	`SELECT`	`TRUE AND NULL`
8	`SELECT`	`FALSE AND NULL`
9	`SELECT`	`(TRUE AND FALSE) OR NULL`

SQL

Operators - Answer

What are the outcomes of the following expressions?

1	`SELECT 4 = NULL`	NULL
2	`SELECT NULL = NULL`	NULL
3	`SELECT NULL IS NULL`	TRUE
4	`SELECT NULL IS NOT NULL`	FALSE
5	`SELECT TRUE OR NULL`	TRUE
6	`SELECT FALSE OR NULL`	NULL
7	`SELECT TRUE AND NULL`	NULL
8	`SELECT FALSE AND NULL`	FALSE
9	`SELECT (TRUE AND FALSE) OR NULL`	NULL

SQL

Operators

Logical Operators

&

AND OR

Example:

```
SELECT      *
FROM        country
WHERE       LifeExpectency > 75 OR Population > 50000000
```

```
SELECT      *
FROM        country
WHERE       IndepYear > 1950 AND GNP > 1000000
```

```
SELECT      *
FROM        country
WHERE       Population < 100000000 AND GNP > 1000000
```

SQL

Operators

Logical Operators

BETWEEN

TRUE if the operand is within the range of comparisons

Example:

```
SELECT        *
FROM          country
WHERE         LifeExpectancy BETWEEN 50 AND 60
```

SQL

Operators

Logical Operators

IN

TRUE if the operand is equal to one of a list of expressions

Example:

```
SELECT       *
FROM         country
WHERE        Continent IN (‘Africa’, ‘Asia’)
```

```
SELECT       *
FROM         country
WHERE        IndepYear IN (1990, 1991, 1992)
```

SQL

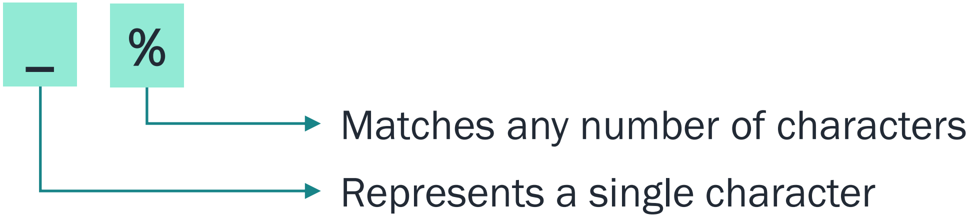
Operators

Logical Operators & Regular Expressions

LIKE

TRUE if the operand matches a pattern

Wild Cards used:



_ Represents a single character

% Matches any number of characters

Example:

```
SELECT       *
FROM         country
WHERE        Continent LIKE 'A%'
```

```
SELECT       *
FROM         country
WHERE        Name LIKE 'C%'
```

SQL

Operators

Logical Operators & Regular Expressions

LIKE

TRUE if the operand matches a pattern

Examples:

column LIKE 'a%'	Finds any values that start with "a"
column LIKE '%a'	Finds any values that end with "a"
column LIKE '%or%'	Finds any values that have "or" in any position
column LIKE '_r%'	Finds any values that have "r" in the second position
column LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
column LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
column LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Aggregation

SQL

Aggregation

Simple Aggregation

SUM MIN MAX AVG STD COUNT

Aggregates all the records of a column by taking the SUM, MIN, MAX, AVG, STD or the number of records.

Example:

```
SELECT        AVG(LifeExpectancy)
FROM          country;
```

```
SELECT        COUNT(*)
FROM          country
WHERE         Continent = ‘Asia’
```

SQL

Aggregation

Group By Aggregation   Used to summarize data based on values of a specific column

```
SELECT        Agg Func (<column name>)
FROM          <table name>
WHERE         <where predicates / conditions>
GROUP BY      <list of columns>
```

Example:

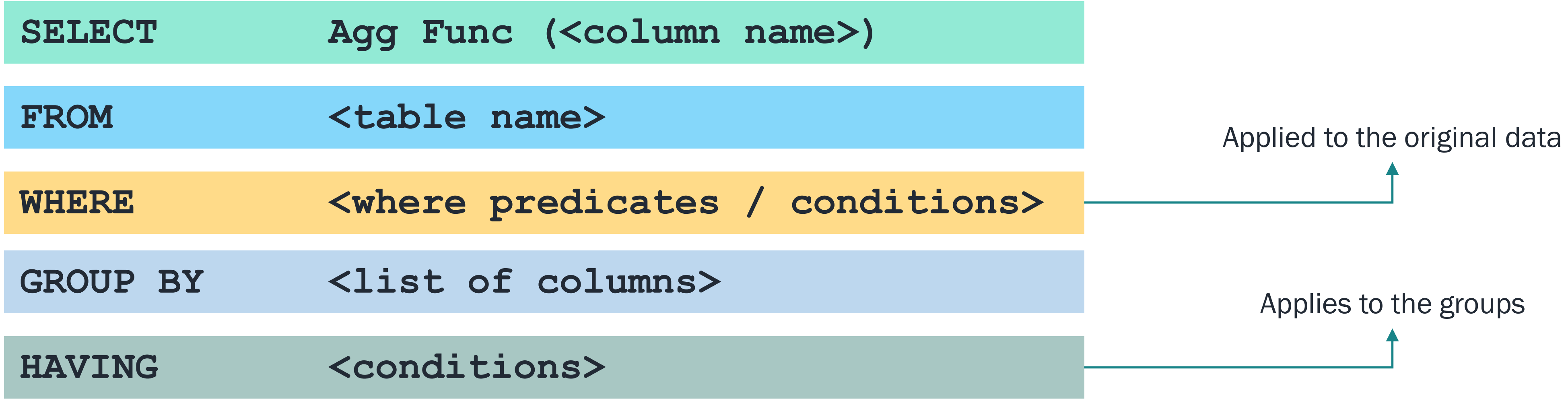
```
SELECT        continent, AVG(LifeExpectancy)
FROM          country
WHERE         Population > 30000
GROUP BY      Continent
```

SQL

Aggregation

Group By Aggregation with Conditions

Used to apply a condition to the groups



```
SELECT        Agg Func (<column name>)
FROM          <table name>
WHERE         <where predicates / conditions>
GROUP BY      <list of columns>
HAVING        <conditions>
```

Applied to the original data

Applies to the groups

Example:

```
SELECT        continent, AVG(LifeExpectancy) AS avg_le
FROM          country
WHERE         Population > 30000
GROUP BY      Continent
HAVING        avg_le > 70
```

Joins

SQL

Joins

Students Table

Student ID	First Name	Last Name	Major	GPA
125	Janet	Logan	EE	3.7
423	Janet	Carroll	CS	3.4
854	Farzad	Kamalzadeh	OR	3.6
239	Alex	Hagen	CE	3.9
371	Janet	Logan	EE	3.8

Courses Table

Course ID	Course Name
6458	Databases
7524	Big Data
6532	Python
4582	ML
3467	Data Mining

Enrollment Table

Course ID	Student ID
6458	423
7524	239
6532	125
4582	371
3467	854

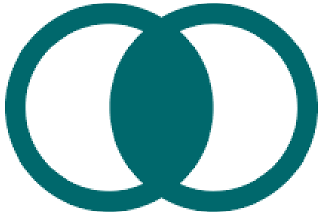
Join of all the above tables

Student ID	First Name	Last Name	Major	GPA	Course ID	Course Name
125	Janet	Logan	EE	3.7	6532	Python
423	Janet	Carroll	CS	3.4	6458	Databases
854	Farzad	Kamalzadeh	OR	3.6	3467	Data Mining
239	Alex	Hagen	CE	3.9	7524	Big Data
371	Janet	Logan	EE	3.8	4582	ML

SQL

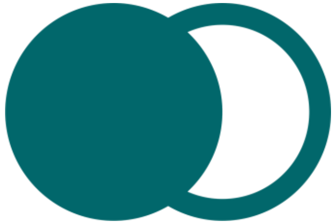
Join Types

Used to combine rows from two or more tables, based on a related column between them



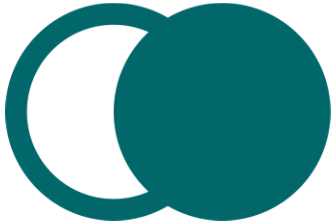
(INNER) JOIN

Returns records that have matching values in **both** tables



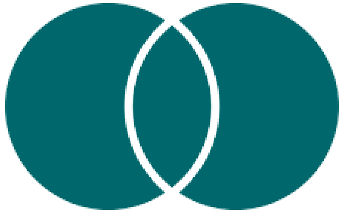
LEFT (OUTER) JOIN

Returns all records from the **left** table, and the matched records from the **right** table



RIGHT (OUTER) JOIN

Returns all records from the **right** table, and the matched records from the **left** table



FULL (OUTER) JOIN

Returns all records when there is a match in **either left or right** table

SQL

Joins

(INNER) JOIN

```
SELECT        <column list>
FROM          <table 1> AS T1, <table 2> AS T2
WHERE         T1.<col 1> = T2.<col 2>
```

Without the condition, it would be Cartesian JOIN, returning all possible combinations

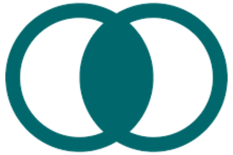
Example:

```
SELECT        *
FROM          country, city
WHERE         country.Code = city.CountryCode
```

SQL

Joins

(INNER) JOIN



```
SELECT        <column list>

FROM          <table 1> AS T1

JOIN          <table 2> AS T2
ON            T1.<col 1> = T2.<col 2>
```

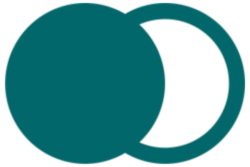
Example:

```
SELECT        *
FROM          customers AS c
JOIN          sales AS s
ON            c.customer_id = s.customer_id
```

SQL

Joins

LEFT (OUTER) JOIN



```
SELECT          <column list>
```

```
FROM            <table 1> AS T1
```

```
LEFT JOIN       <table 2> AS T2
ON              T1.<col 1> = T2.<col 2>
```

Returns all records from the left table (table 1), and the matching records from the right table (table 2)

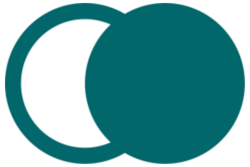
Example:

```
SELECT          *
FROM            customers AS c
LEFT JOIN       sales AS s
ON              c.customer_id = s.customer_id
```

SQL

Joins

RIGHT (OUTER) JOIN



```
SELECT        <column list>
```

```
FROM          <table 1> AS T1
```

```
RIGHT JOIN    <table 2> AS T2
ON            T1.<col 1> = T2.<col 2>
```

Returns all records from the right table (table 2), and the matching records from the left table (table 1)

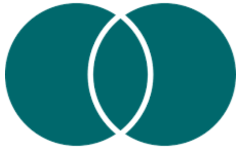
Example:

```
SELECT        *
FROM          customers AS c
RIGHT JOIN    sales AS s
ON            c.customer_id = s.customer_id
```

SQL

Joins

FULL (OUTER) JOIN



```
SELECT        <column list>

FROM          <table 1> AS T1

CROSS JOIN    <table 2> AS T2
ON            T1.<col 1> = T2.<col 2>
```

Example:

```
SELECT        *
FROM          customers AS c
CROSS JOIN    sales AS s
ON            c.customer_id = s.customer_id
```

SQL

Exercise

1. Exercise1

```
Table: Person

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| personId    | int     |
| lastName    | varchar |
| firstName   | varchar |
+-------------+---------+
personId is the primary
key    column    for    this
table.
This      table      contains
information about the ID
of some persons and their
first and last names.
```

```
Table: Address

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| addressId   | int     |
| personId    | int     |
| city        | varchar |
| state       | varchar |
+-------------+---------+
addressId is the primary
key    column    for    this
table.
Each  row  of  this  table
contains       information
about the city and state
of  one  person  with  ID  =
PersonId.
```

Write an SQL query to report the first name, last name, city, and state of each person in the Person table. If the address of a personId is not present in the Address table, report null instead.

Return the result table in any order.

```
Person table:

+----------+----------+-----------+
| personId | lastName | firstName |
+----------+----------+-----------+
| 1        | Wang     | Allen     |
| 2        | Alice    | Bob       |
+----------+----------+-----------+
```

```
Address table:

+-----------+----------+---------------+------------+
| addressId | personId | city          | state      |
+-----------+----------+---------------+------------+
| 1         | 2        | New York City | New York   |
| 2         | 3        | Apple         | California |
+-----------+----------+---------------+------------+
```

```
Output:
+-----------+----------+---------------+----------+
| firstName | lastName | city          | state    |
+-----------+----------+---------------+----------+
| Allen     | Wang     | Null          | Null     |
| Bob       | Alice    | New York City | New York |
+-----------+----------+---------------+----------+
```

SQL

Answer

```
Person table:

+----------+----------+-----------+
| personId | lastName | firstName |
+----------+----------+-----------+
| 1        | Wang     | Allen     |
| 2        | Alice    | Bob       |
+----------+----------+-----------+
```

```
Address table:

+-----------+----------+---------------+------------+
| addressId | personId | city          | state      |
+-----------+----------+---------------+------------+
| 1         | 2        | New York City | New York   |
| 2         | 3        | Leetcode      | California |
+-----------+----------+---------------+------------+
```

```sql
SELECT          P.firstname, P.lastname, A.city, A.state
FROM            Person AS P
LEFT JOIN       Address AS A
ON              P.personId = A.personId
```

```
Output:
+-----------+----------+---------------+----------+
| firstName | lastName | city          | state    |
+-----------+----------+---------------+----------+
| Allen     | Wang     | Null          | Null     |
| Bob       | Alice    | New York City | New York |
+-----------+----------+---------------+----------+
```

SQL

Exercise

2. Exercise2

```
Table: Employee

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| salary      | int     |
| managerId   | int     |
+-------------+---------+
```

id is the primary key column for this table. Each row of this table indicates the ID of an employee, their name, salary, and the ID of their manager.

```
Employee table:

+----+-------+--------+-----------+
| id | name  | salary | managerId |
+----+-------+--------+-----------+
| 1  | Joe   | 70000  | 3         |
| 2  | Henry | 80000  | 4         |
| 3  | Sam   | 60000  | Null      |
| 4  | Max   | 90000  | Null      |
+----+-------+--------+-----------+
```

Write an SQL query to find the employees who earn more than their managers. Return the result table in **any order.**

```
Output:
+----------+
| Employee |
+----------+
| Joe      |
+----------+
```

SQL

Answer

```
Employee table:

+----+-------+--------+-----------+
| id | name  | salary | managerId |
+----+-------+--------+-----------+
| 1  | Joe   | 70000  | 3         |
| 2  | Henry | 80000  | 4         |
| 3  | Sam   | 60000  | Null      |
| 4  | Max   | 90000  | Null      |
+----+-------+--------+-----------+
```

```
SELECT      a.Name AS 'Employee'
FROM        Employee AS a,
            Employee AS b
WHERE       a.ManagerId = b.Id
AND         a.Salary > b.Salary
```

```
SELECT      a.NAME AS Employee
FROM        Employee AS a
JOIN        Employee AS b
ON          a.ManagerId = b.Id
AND         a.Salary > b.Salary
```

```
Output:
+----------+
| Employee |
+----------+
| Joe      |
+----------+
```

SQL

Exercise

3. Exercise3

Table: Customers

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
+-------------+---------+
```

id is the primary key column for this table. Each row of this table indicates the ID and name of a customer.

Table: Orders

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| id          | int  |
| customerId  | int  |
+-------------+------+
```

id is the primary key column for this table. customerId is a foreign key of the ID from the Customers table. Each row of this table indicates the ID of an order and the ID of the customer who ordered it.

Write an SQL query to find the customer who did not order.
Return the result table in **any order**.

```
Customers table:
+----+-------+
| id | name  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+
```

```
Orders table:
+----+------------+
| id | customerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

```
Output:
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

SQL

Answer

```
Customers table:
+----+-------+
| id | name  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+
```

```
Orders table:
+----+------------+
| id | customerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

```
SELECT         Name AS 'Customers'
FROM           Customers c
LEFT JOIN      Orders o
ON             c.Id = o.CustomerId
WHERE          o.CustomerId IS NULL
```

```
Output:
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

SQL

Exercise

4. Exercise4

```
Table: Person

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| email       | varchar |
+-------------+---------+
id is the primary key
column for this table.
Each row of this table
contains an email. The
emails will not contain
uppercase letters.
```

Write an SQL query to report all the duplicate emails.

Return the result table in any order.

```
Person table:
+----+---------+
| id | email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

```
Output:
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

SQL

Answer

```
Person table:
+----+---------+
| id | email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

```
SELECT      Email
FROM        Person
GROUP BY    Email
HAVING      count(Email) > 1
```

```
Output:
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

Sub-Queries

SQL

Sub-Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the SELECT clause (column Expression)

```
SELECT          <list of columns>, (sub-query)

FROM            <table name>
```

Example:

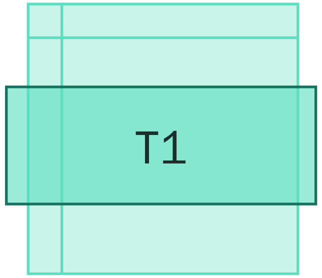
```
SELECT          emp_id, salary
                (SELECT AVG(salary) AS avg_sal)
                 FROM employees
FROM            employees
```

SQL

Sub-Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the FROM clause (table expression)

```
SELECT          <list of columns>
FROM            (Sub-Query)AS T1
```



Example:

```
SELECT          *
FROM            (SELECT         *
                 FROM           country
                 WHERE continent = ‘Asia’) AS T1
LIMIT           10
```

SQL

Sub-Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause

```
SELECT       <list of columns>
FROM         <table name>
WHERE        <condition with sub-query>
```

Example:

```
SELECT       *
FROM         country
WHERE        continent IN (SELECT continent
                           FROM   country
                           GROUP BY continent)
```

SQL

Exercise

5. Exercise5

```
Table: Customers

+--------------+----------+
| Column Name | Type     |
+--------------+----------+
| id           | int      |
| name         | varchar |
+--------------+----------+
id is the primary key
column for this table.
Each row of this table
indicates the ID and name
of a customer.
```

```
Table: Orders

+--------------+-------+
| Column Name | Type |
+--------------+-------+
| id           | int   |
| customerId   | int   |
+--------------+-------+
id is the primary key
column for this table.
customerId is a
foreign key of the ID
from the Customers
table.
Each row of this table
indicates the ID of an
order and the ID of
the customer who
ordered it.
```

Write an SQL query to find the customer who did not order.
Return the result table in **any order.**

```
Customers table:
+----+-------+
| id | name  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+
```

```
Orders table:
+----+------------+
| id | customerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

```
Output:
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

SQL

Answer

```
Customers table:
+----+--------+
| id | name   |
+----+--------+
| 1  | Joe    |
| 2  | Henry  |
| 3  | Sam    |
| 4  | Max    |
+----+--------+
```

```
Orders table:
+----+------------+
| id | customerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

```
SELECT          customers.name as 'Customers'
FROM            customers
WHERE           customers.id NOT IN
                      (SELECT      customerId
                       FROM        orders)
```

```
Output:
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

SQL

Exercise

6. Exercise6

```
Table: Person

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| email       | varchar |
+-------------+---------+
```

id is the primary key column for this table. Each row of this table contains an email. The emails will not contain uppercase letters.

Write an SQL query to report all the duplicate emails.

Return the result table in any order.

```
Person table:
+----+---------+
| id | email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

```
Output:
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

SQL

Answer

```
Person table:
+-----+---------+
| id | email    |
+-----+---------+
| 1   | a@b.com |
| 2   | c@d.com |
| 3   | a@b.com |
+-----+---------+
```

```
SELECT          Email
FROM            (SELECT Email, count(Email) as num
                 FROM Person
                 GROUP BY Email) AS T1
WHERE           num > 1
```

```
Output:
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

SQL

More with Sub-Queries

To check sub-queries

EXISTS	To check if sub-query result is empty or not
ANY	To check if condition holds for some sub-query rows
ALL	To check if condition holds for all sub-query rows

Syntax:

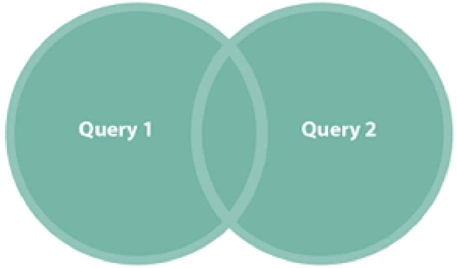
`EXISTS            (<sub_query>)`	TRUE if the sub-query is non-empty
`<value> >= ANY(<sub_query>)`	TRUE if satisfied for some rows
`<value> >= ALL(<sub_query>)`	TRUE if satisfied for all rows

Set Operators

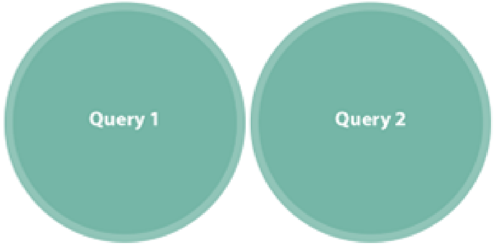
SQL

Set Operations

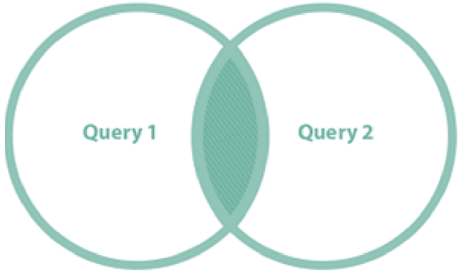
Used to combine, intersect or subtract rows from two or more tables



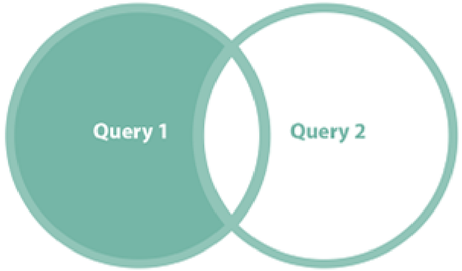
UNION



UNION ALL



INTERSECT



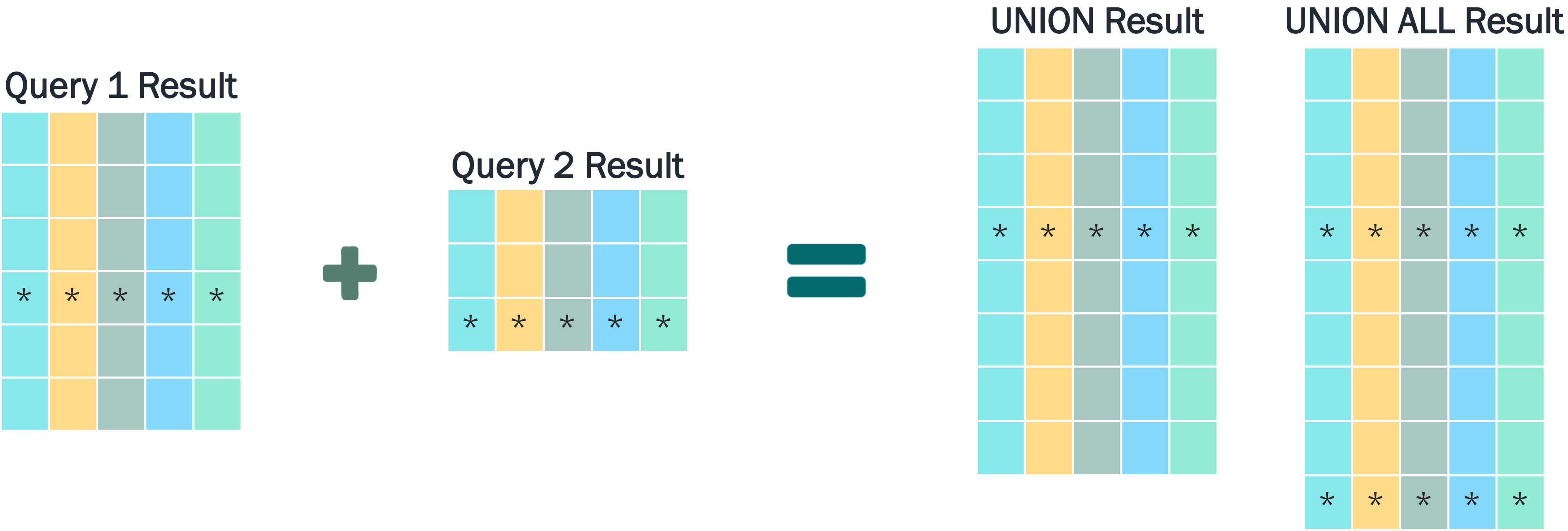
EXCEPT
(NOT IN)

NOTE: the queries must be compatible (have the same columns)

SQL

Set Operations

- UNION (ALL)



Query 1 Result

Query 2 Result

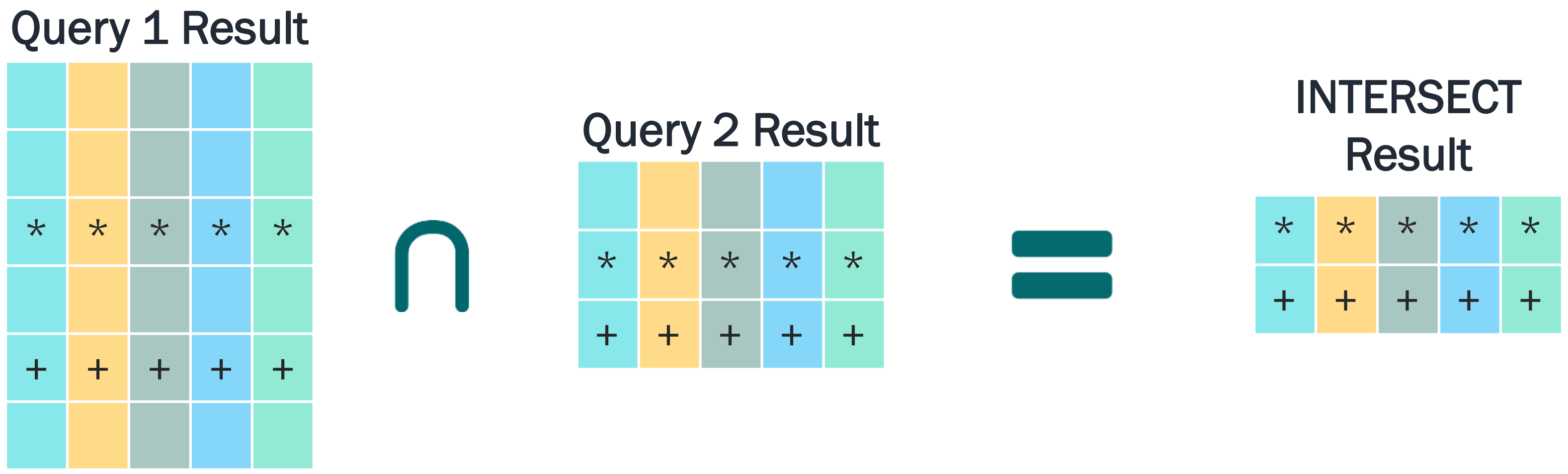
UNION Result

UNION ALL Result

SQL

Set Operations

INTERSECT



Query 1 Result

∩

Query 2 Result

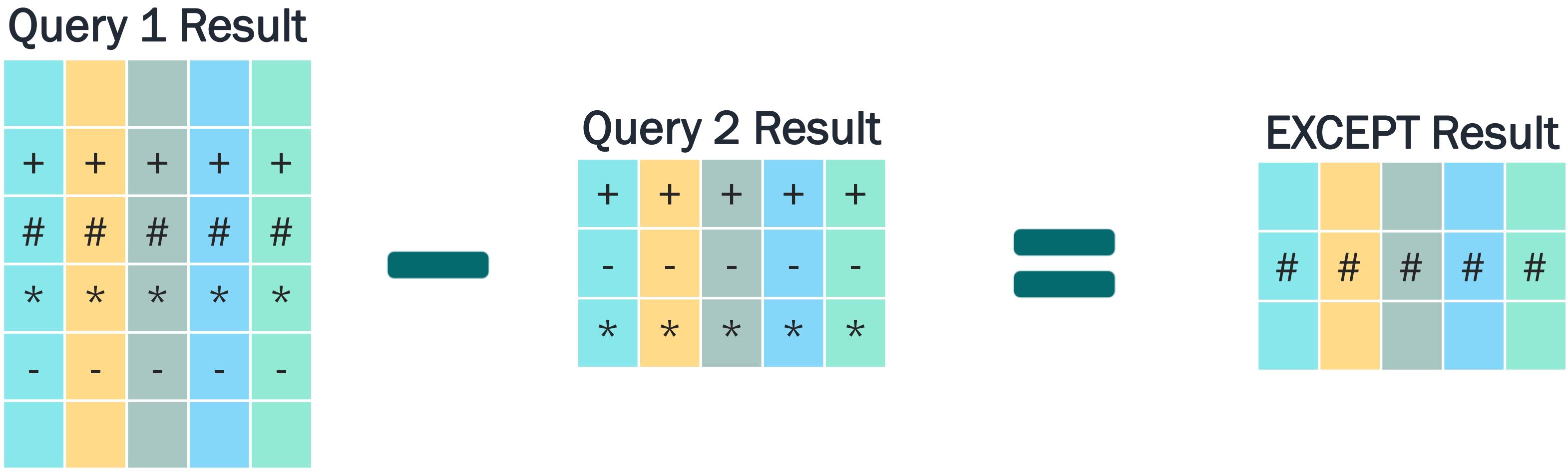
=

INTERSECT Result

SQL

Set Operations

- EXCEPT (NOT IN)



Query 1 Result

+	+	+	+	+
#	#	#	#	#
*	*	*	*	*
-	-	-	-	-

Query 2 Result

+	+	+	+	+
-	-	-	-	-
*	*	*	*	*

EXCEPT Result

#	#	#	#	#

SQL

Set Operations

UNION (ALL), INTERSECT, EXCEPT

```
<query 1>
UNION/INTERSECT/EXCEPT
<query 2>
```

Example:

```
SELECT        a,b    FROM table1
UNION
SELECT        a,b    FROM table2
```

DDL

SQL

DDL

DDL
Data Definition Language

Define admissible database content (schema)

DDL
(Data
Definition
Language)

CREATE
DROP
ALTER
RENAME

Relations and their Schemata
What columns each table have and what are the column types

Constraints to restrict admissible contents
Constraints on single relations and constraints linking multiple relations

SQL

DDL: Table Creation

Table Creation

To create a table, we need to know the following:

- Schema
- Table Name
- Column Names
- Data Types
- Allow Duplicates
- Allow Nulls
- Constraints

SQL

DDL: Table Creation

Table Creation Syntax

```
CREATE TABLE  <table name>
              ( <table definition> )
```

<table definition> is a comma-separated column definition: **<column name> <column type>**

Example:

```
CREATE TABLE  students
              (student_id   INT,
               first_name   VARCHAR(20),
               last_name    VARCHAR(20))
```

SQL

DDL: Data Types

Data types (feature/column types)

Character String

Fixed length: `CHAR()`

Variable Length: `VARCHAR()`

Numeric

Integer: `INT(), SMALINT, BIGINT`

Decimal:

```
DECIMAL(n,m)
DOUBLE
FLOAT
REAL(n,m)
```

Boolean

True/False: `BOOLEAN`

Date/Time

Date: `DATE`

Time: `TIME()`

Date & Time: `DATETIME()`

Time Stamp: `TIMESTAMP()`

SQL

DDL: Data Types

Character String

Fixed length: `CHAR()`

Variable Length: `VARCHAR()`

Value	CHAR(4)	Storage Required	VARCHAR(4)	Storage Required
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes

SQL

DDL: Data Types

Numeric

Integer: `INT(), SMALINT, BIGINT`

Type	Storage (Bytes)	Minimum Value Signed	Minimum Value Unsigned	Maximum Value Signed	Maximum Value Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	-8388608	0	8388607	16777215
INT	4	-2147483648	0	2147483647	4294967295
BIGINT	8	-2^{63}	0	$2^{63}-1$	$2^{64}-1$

SQL

DDL: Data Types

Numeric

Decimal:

```
DECIMAL(n,m)
DOUBLE
FLOAT
REAL(n,m)
```

Fixed-Point Types
(Exact Value)
- DECIMAL, NUMERIC

Floating-Point Types
(Approximate Value)
- FLOAT, DOUBLE

SQL

DDL: Data Types

Numeric:
Fixed-Point Types (Exact Value) - DECIMAL, NUMERIC

- The DECIMAL and NUMERIC types store exact numeric data values. These types are used when it is important to preserve exact precision, for example with monetary data.
- The maximum number of digits for DECIMAL is 65, but the actual range for a given DECIMAL column can be constrained by the precision or scale for a given column. When such a column is assigned a value with more digits following the decimal point than are permitted by the specified scale, the value is converted to that scale. (The precise behavior is operating system-specific, but generally the effect is truncation to the permissible number of digits.)

Floating-Point Types (Approximate Value) - FLOAT, DOUBLE

- The FLOAT and DOUBLE types represent approximate numeric data values. MySQL uses four bytes for single-precision values and eight bytes for double-precision values.
- Because floating-point values are approximate and not stored as exact values, attempts to treat them as exact in comparisons may lead to problems. They are also subject to platform or implementation dependencies.

- https://stackoverflow.com/questions/1056323/difference-between-numeric-float-and-decimal-in-sql-server

SQL

DDL: Data Types

Date/Time

Date: `DATE`

Time: `TIME(fsp)`

Date & Time: `DATETIME(fsp)`

Time Stamp: `TIMESTAMP(fsp)`

The *fsp* value, if given, must be in the range 0 to 6. A value of 0 signifies that there is no fractional part. If omitted, the default precision is 0.

```
CREATE TABLE t1
                (t TIME(3),
                 dt DATETIME(6),
                 ts TIMESTAMP(0));
```

Data Type	Example	Format
DATE	'1992-02-10'	*'YYYY-MM-DD'*
TIME	'10:35:24'	*hh:mm:ss[.fraction]'*
DATETIME	'1992-02-10 10:35:24'	*'YYYY-MM-DD hh:mm:ss[.fraction]'*
TIMESTAMP	'1992-02-10 10:35:24'	*'YYYY-MM-DD hh:mm:ss[.fraction]'*
YEAR	1992	*YYYY*

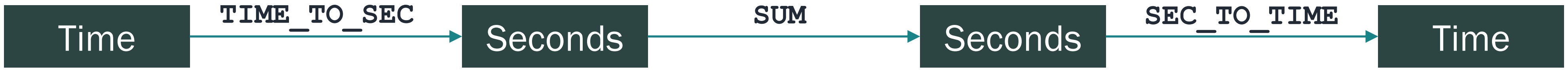
SQL

DDL: Data Types

Date/Time

Date:	`DATE`
Time:	`TIME()`
Date & Time:	`DATETIME()`
Time Stamp:	`TIMESTAMP()`

- The SUM() and AVG() aggregate functions do not work with temporal values.
  - (They convert the values to numbers, losing everything after the first nonnumeric character.)
- To work around this problem, convert to numeric units, perform the aggregate operation, and convert back to a temporal value.



```
SELECT       SEC_TO_TIME(  SUM(   TIME_TO_SEC( time_col)    ))
FROM         tbl_name;

SELECT       FROM_DAYS(    SUM(   TO_DAYS(     date_col)    ))
FROM         tbl_name;
```

SQL

DDL: Data Types

CAST Function

The CAST() function in MySQL is used to convert a value from one data type to another data type specified in the expression.

```
CAST(<expression> AS <datatype>)
```

- Expression: It is a value that will be converted into another specific datatype.
- Datatype: It is a value or data type in which the expression value needs to be converted.

Example:

```
SELECT       CAST("2018-11-30" AS DATE);
SELECT       CAST(3-6 AS SIGNED);
SELECT       CONCAT('CAST Function Example ## ',CAST(5 AS CHAR));
```

SQL

DDL: Data Types

Date/Time Functions

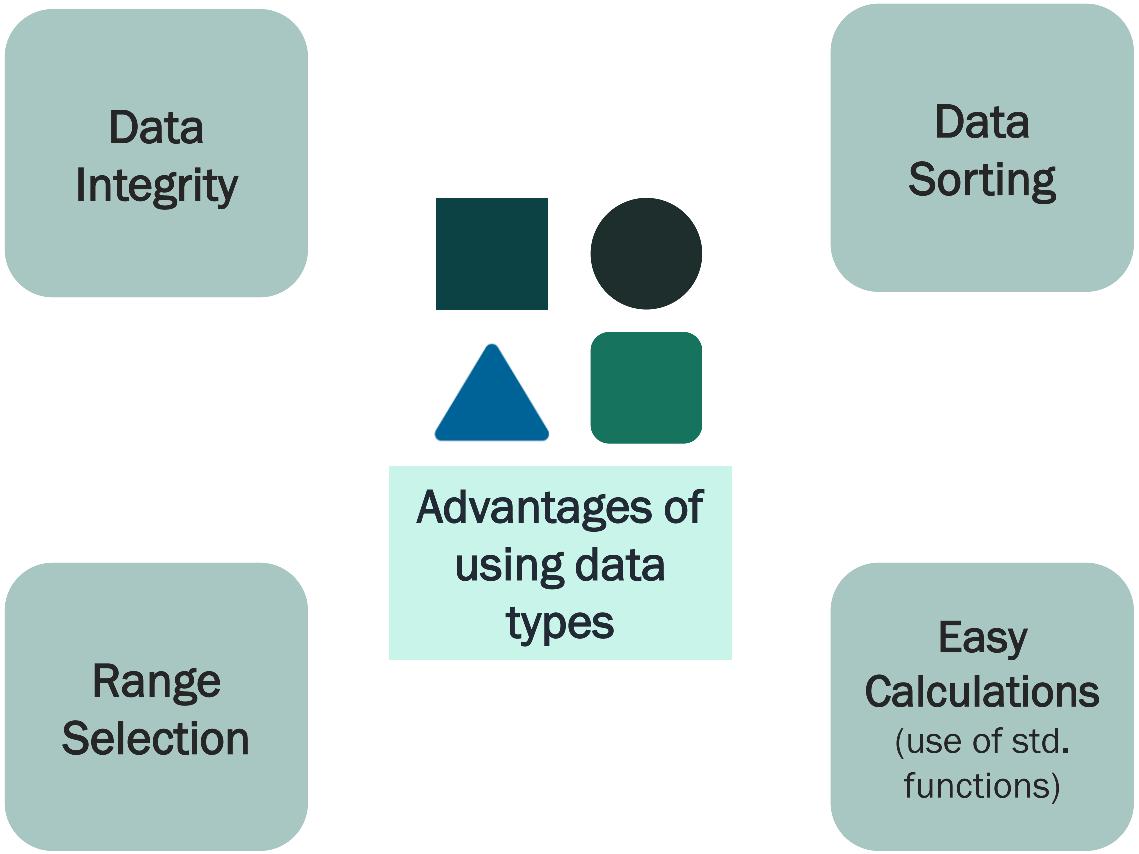
Below is a list of useful date and time functions in MySQL

(A complete List of these functions)

Name	Description
CURDATE()	Return the current date
CURTIME()	Return the current time
DATE_FORMAT()	Format date as specified
DATEDIFF()	Subtract two dates
NOW()	Return the current date and time
STR_TO_DATE()	Convert a string to a date
TIME_FORMAT()	Format as time
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPDIFF()	Subtract an interval from a datetime expression

SQL

DDL: Data Types



Advantages of using data types

Data Integrity

Data Sorting

Range Selection

Easy Calculations (use of std. functions)

SQL

DDL: Primary Key Constraint

Table Creation Syntax with Primary Key

```
CREATE TABLE <table name>
             ( <table definition>,
              PRIMARY KEY (<column name>) )
```

```
CREATE TABLE <table name>
             ( <table definition>,
              CONSTRAINT <constraint name>
              PRIMARY KEY (<column names>) )
```

Example:

```
CREATE TABLE students
             (student_id   INT,
              first_name   VARCHAR(20),
              last_name    VARCHAR(20),
              PRIMARY KEY (student_id))
```

SQL

DDL: Foreign Key Constraint

Table Creation Syntax with Foreign Key

```
CREATE TABLE <table name>
             ( <table definition>,
              CONSTRAINT <constraint name>
              FOREIGN KEY (<column name>, ...)
              REFERENCES <tbl_name> (<col_name>,...)
              ON DELETE <reference_option>
              ON UPDATE <reference_option>)
```

```
reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

Example:

```
CREATE TABLE     enrollment
                 (student_id      INT,
                  first_name      VARCHAR(20),
                  last_name       VARCHAR(20),
                  CONSTRAINT fk1
                         FOREIGN KEY (student_id)
                         REFERENCES students(std_id)
                         ON DELETE CASCADE
                         ON UPDATE NO ACTION
```

SQL

DDL

- Referential Actions
- When an UPDATE or DELETE operation affects a key value in the parent table that has matching rows in the child table, the result depends on the referential action specified by ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause. Referential actions include:
- CASCADE: Delete or update the row from the parent table and automatically delete or update the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported. Between two tables, do not define several ON UPDATE CASCADE clauses that act on the same column in the parent table or in the child table.
- If a FOREIGN KEY clause is defined on both tables in a foreign key relationship, making both tables a parent and child, an ON UPDATE CASCADE or ON DELETE CASCADE subclause defined for one FOREIGN KEY clause must be defined for the other in order for cascading operations to succeed. If an ON UPDATE CASCADE or ON DELETE CASCADE subclause is only defined for one FOREIGN KEY clause, cascading operations fail with an error.
- SET NULL: Delete or update the row from the parent table and set the foreign key column or columns in the child table to NULL. Both ON DELETE SET NULL and ON UPDATE SET NULL clauses are supported.
- If you specify a SET NULL action, make sure that you have not declared the columns in the child table as NOT NULL.
- RESTRICT: Rejects the delete or update operation for the parent table. Specifying RESTRICT (or NO ACTION) is the same as omitting the ON DELETE or ON UPDATE clause.
- NO ACTION: A keyword from standard SQL. In MySQL, equivalent to RESTRICT. The MySQL Server rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table. Some database systems have deferred checks, and NO ACTION is a deferred check. In MySQL, foreign key constraints are checked immediately, so NO ACTION is the same as RESTRICT.
- SET DEFAULT: This action is recognized by the MySQL parser, but both InnoDB and NDB reject table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET DEFAULT clauses.

SQL

DDL: Alteration

Alteration

Alter is used for:

- Add/remove columns
- Add/remove keys
- Modify the data types
- Add/remove constraints

SQL

DDL: Alteration

Add columns

```
ALTER TABLE          <table name>
ADD COLUMN           <column name> <data type>
                     <other options>
```

```
Other options:
       NULL | NOT NULL | FIRST | AFTER <column name> | BINARY | …
```

Example:

```
ALTER TABLE   students
ADD COLUMN    age INT NULL AFTER last_name
```

SQL

DDL: Alteration

Remove columns

```
ALTER TABLE         <table name>
DROP COLUMN         <column name>
```

Example:

```
ALTER TABLE  students
DROP COLUMN  age
```

SQL

DDL: Alteration

Change/modify/rename columns

```
ALTER TABLE           <table name>
CHANGE COLUMN         <column name>
                      <new column name> <new definition>
```

```
ALTER TABLE           <table name>
MODIFY                <column name> <new definition>
```

Example:

```
ALTER TABLE           students
CHANGE COLUMN         last_name
                      lname VARCHAR(20) NOT NULL
```

SQL

DDL: Alteration

Rename table

```
ALTER TABLE            <table name>
RENAME TO              <new table name>
```

Example:

```
ALTER TABLE            students
RENAME TO              students_table
```

SQL

DDL: Alteration

Add Primary Key constraint

```
ALTER TABLE         <table name>
ADD PRIMARY KEY     (<column name>)
```

```
ALTER TABLE         <table name>
ADD CONSTRAINT      <constraint name>
PRIMARY KEY         (<column name>)
```

Example:

```
ALTER TABLE         students
ADD PRIMARY KEY     (student_id)
```

SQL

DDL: Alteration

Drop Primary Key constraint

```
ALTER TABLE         <table name>
DROP PRIMARY KEY
```

Example:

```
ALTER TABLE         students
DROP PRIMARY KEY
```

SQL

DDL: Alteration

Add Foreign Key constraint

```
ALTER TABLE          <table name>
ADD FOREIGN KEY      (<column name>)
REFERENCES           <reference table>(<reference column>)
```

```
ALTER TABLE          <table name>
ADD CONSTRAINT       <constraint name>
FOREIGN KEY          (<column name>)
REFERENCES           <reference table>(<reference column>)
```

Example:

```
ALTER TABLE          enrollment
ADD CONSTRAINT       fk_enrol_std
FOREIGN KEY          (student_id)
REFERENCES           students(std_id)
```

DML

SQL

DML

DML
Data Manipulation Language

Used to manipulate the data of an existing schema

DML
(Data Manipulation Language)

INSERT
UPDATE
DELETE

Adding/inserting data into a table

Deleting/removing data from a table

Updating existing data from a table

SQL

DML: Data Insertion

Data Insertion into a table

DML is used to insert data into an existing table

Add a complete row
(fully specified row)

Add a part of a row
(partially specified row)

Add data from a file

SQL

DML: Insertion

Fully specified row

```
INSERT INTO             <table name>
VALUES                  (<list of values>)
```

```
INSERT INTO             <table name>
VALUES                  (<value list 1>),
                        (<value list 2>),
                        (<value list 3>), …
```

Values should be separated by comma and in the same order as the columns

Example:

```
INSERT INTO             students
VALUES                  (765427, ‘Alex’, ‘Hagen’)
```

SQL

DML: Insertion

Partially specified row

```
INSERT INTO              <table name> (<list of columns>)
VALUES                   (<list of values>)
```

Values should be separated by comma and in the same order as the columns

Example:

```
INSERT INTO              students (std_id, first_name)
VALUES                   (765427, ‘Alex’)
```

SQL

DML: Insertion

From a file

```
LOAD DATA (LOCAL) INFILE   ‘<location of the file>’
INTO TABLE                 <table name>
FIELDS TERMINATED BY       ‘<field delimiter>'
ENCLOSED BY                ‘<string identifier>'
LINES TERMINATED BY        ‘<new line identifier>'
IGNORE 1 ROWS;
```

First row can be ignored if it contains the header (names of the columns)

https://dev.mysql.com/doc/refman/5.7/en/load-data.html

Example:

```
LOAD DATA INFILE           ‘c:/temp/students.csv’
INTO TABLE                 students
FIELDS TERMINATED BY       ‘,’
ENCLOSED BY                ‘”’
LINES TERMINATED BY        ‘\n’
IGNORE 1 ROWS;
```

https://stackoverflow.com/questions/32737478/how-should-i-resolve-secure-file-priv-in-mysql

SQL

DML: Data Deletion

Deleting data from a table

DML is used to delete data from an existing table

```
DELETE FROM          <table name>
WHERE                <condition>
```

Example:

```
DELETE FROM          students
WHERE                std_id = 126417
```

SQL

DML: Data Update

Updating data in a table

DML is also used to update data in an existing table

```
UPDATE              <table name>
SET                 <column name> = <value>
WHERE               <condition>
```

Example:

```
UPDATE              students
SET                 last_name = ‘Hagen’
WHERE               std_id = 765427
```

Indexes

SQL

Indexes

How to create an index

```
CREATE INDEX        <index name>
ON                  <table name> (<list of columns>)
```

List of columns is the column names separated by commas.

Example:

```
CREATE INDEX        emp_name_idx
ON                  employees (emp_first_name)
```

```
CREATE INDEX        emp_name_idx
ON                  employees (emp_first_name(10))
```

SQL

Indexes

How to see all the indexes from a table

```
SHOW INDEX
FROM                    <table name>
FROM                    <database name>
```

Lists all the indexes from a given table in a given database

Example:

```
SHOW INDEX
FROM                    students
FROM                    university_db
```

Transactions

SQL

Transactions

How to create a transaction

```
START TRANSACTION;

<query>;
<query>;
…
<query>;

COMMIT;
```

Options:

`SET autocommit = 0;`	Disables the autocommit option
`ROLLBACK`	Rolls back the changes made instead of committing them

Query Timing

SQL

Query Timing

How to time your queries

```
SET PROFILING = 1;

<query 1>;
<query 2>;
…
<query n>;

SHOW PROFILES;
```

Show profile will return a table of the queries executed and the time it took for them to run

MQL

MQL

Basic Commands

```
show dbs
```

Show all the databases

```
use <db name>
```

Switches to the given db, creates one if it does not exist

```
db
```

Shows the current db being worked on

```
show collections
```

Shows all the collections in the current db

https://www.mongodb.com/developer/quickstart/cheat-sheet/

MQL

Query

```
db.<collection name>.find()
```

Shows all the documents in the collection

```
db.<collection name>.find().pretty()
```

Show all the docs in the coll, formatted

```
db.<collection name>.findOne()
```

Returns a single document

```
db.<collection name>.distinct(“<property>”)
```

Returns all the distinct values the given property has in the database

MQL

Query

```
db.<collection name>.find({specify a condition})
```

Shows all the matches from the database

```
db.<collection name>.find({$or:[{condition1},{condition2}]})
```

Shows all the matches from the database given the list of conditions

Example:

```
db.students.find({name:”Max”})
```

```
db.students.find({date: ISODate("2020-09-25")})
```

```
db.students.find({$and:[{name:”Max”},{major:”EE”}]})
```

```
db.students.find({name: "Max", age: 32})
```

MQL

Query - Counts

```
db.<collection name>.count()
```

Counts the number of documents that match the criteria (based on collection metadata)

```
db.<collection name>.countDocuments()
```

Counts the number of documents that match the criteria (accurate count)

Example:

```
db.students.count({age: 32})
```

```
db.students.countDocuments({age: 32})
```

MQL

Comparison Operators

Comparison Operators

$eq $ne $gt $lt $gte $lte $in $nin

Used to compare the values of two operands

Example:

```
db.students.find({"year": {$gt: 1970}})
db.students.find({"year": {$gte: 1970}})
db.students.find({"year": {$lt: 1970}})
db.students.find({"year": {$lte: 1970}})
db.students.find({"year": {$ne: 1970}})
db.students.find({"year": {$in: [1958, 1959]}})
db.students.find({"year": {$nin: [1958, 1959]}})
```

MQL

Logical Operators

Logical Operators

$not $or $nor $and

Used to combine logical operations

Example:

```
db.students.find({name:{$not: {$eq: "Max"}}})
db.students.find({$or: [{"year" : 1958}, {"year" : 1959}]})
db.students.find({$nor: [{gpa: 1.99}, {failed: true}]})
db.students.find({
  $and: [
    {$or: [{gpa: {$lt :3}}, {gpa :{$gt: 2}}]},
    {$or: [{failed: true}, {gpa: {$lt: 3 }}]}
  ]
})
```

MQL

Element Operators

Element Operators

$exists	Used to check if a specific property exists
$type	Used for checking the value types in the database https://www.mongodb.com/docs/manual/reference/operator/query/type/

Example:

```
db.students.find({name: {$exists: true}})
db.students.find({"zipCode": {$type: 2 }})
db.students.find({"zipCode": {$type: "string"}})
```

MQL

Create

```
db.createCollection(‘<collection name>’)
```

Creates a collection

```
db.<collection name>.insertOne()
```

Inserts a new document into the collection

Example:

```
db.createCollection(‘students’)
```

```
db.students.insertOne({name: "Max"})
```

MQL

Create

```
db.<collection name>.insert()
```

Inserts a new document into the collection

```
db.<collection name>.insert([,])
```

Inserts multiple documents into the collection

Example:

```
db.students.insert([{name: "Max"}, {name:"Alex"}])
```

```
db.students.insert({date: new Date(‘2021-11-21’)})
```

https://www.mongodb.com/docs/manual/reference/method/Date/

MQL

Update

```
db.<collection name>.update({<condition>},{update})
```

replaces an existing document with the new fields and values

```
db.<collection name>.update({<condition>}, {$set:{update}})
```

Updates an existing document keeping the rest of the properties untouched

https://www.mongodb.com/docs/manual/reference/method/db.collection.update/

Example:

```
db.students.update({"_id": 1}, {"year": 2016})
```

```
db.students.update({"_id": 1}, {$set: {"year": 2016, name:
"Max"}})
```

MQL

Update

```
db.<collection name>.update({<condition>},{$unset:{update}})
```

Removes a property

```
db.<collection name>.update({<condition>}, {$rename:{“<f>”:”<g>”}})
```

Renames field (property) f to g

Example:

```
db.students.update({"_id": 1}, {$unset: {"year": 1}})
```

```
db.students.update({"_id": 1}, {$rename: {"year": "date"} })
```

MQL

Delete

```
db.<collection name>.remove({<condition>})
```

Removes documents based on the condition

```
db.<collection name>.findOneAndDelete({<condition>})
```

Remove one document based on the condition

Example:

```
db.students.remove({name: "Max"})
```

```
db.students.findOneAndDelete({"name": "Max"})
```

https://www.mongodb.com/docs/manual/tutorial/query-embedded-documents/

Exercises

SQL

Exercise

197. Rising Temperature

```
Table: Weather

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| recordDate    | date    |
| temperature   | int     |
+---------------+---------+
```

id is the primary key for this table.
This table contains information about the temperature on a certain day.

```
Weather table:
+----+------------+-------------+
| id | recordDate | temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
+----+------------+-------------+
```

Write an SQL query to find all dates' Id with higher temperatures compared to its previous dates (yesterday).

Return the result table in any order.

```
Output:
+----+
| id |
+----+
| 2  |
| 4  |
+----+
```

SQL

Answer

```
Weather table:
+----+------------+-------------+
| id | recordDate | temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
+----+------------+-------------+
```

```sql
SELECT         weather.id AS 'Id'
FROM           weather
JOIN           weather w
ON             DATEDIFF(weather.recordDate, w.recordDate) = 1
               AND weather.Temperature > w.Temperature;
```

```
Output:
+----+
| id |
+----+
| 2  |
| 4  |
+----+
```

SQL

Exercise

603. Consecutive Available Seats

```
Table: Cinema

+-------------+------+
| Column Name | Type |
+-------------+------+
| seat_id     | int  |
| free        | bool |
+-------------+------+
```

seat_id is an auto-increment primary key column for this table. Each row of this table indicates whether the ith seat is free or not. 1 means free while 0 means occupied.

```
Cinema table:
+---------+------+
| seat_id | free |
+---------+------+
| 1       | 1    |
| 2       | 0    |
| 3       | 1    |
| 4       | 1    |
| 5       | 1    |
+---------+------+
```

Write an SQL query to report all the consecutive available seats in the cinema.

Return the result table ordered by seat_id in ascending order.

```
Output:
+---------+
| seat_id |
+---------+
| 3       |
| 4       |
| 5       |
+---------+
```

SQL

Answer

```
SELECT          a.seat_id, a.free, b.seat_id, b.free
FROM            cinema a
JOIN            cinema b;
```

```
SELECT          a.seat_id, a.free, b.seat_id, b.free
FROM            cinema a
JOIN            cinema b
ON              ABS(a.seat_id - b.seat_id) = 1
                AND a.free = true and b.free = true;
```

```
SELECT          distinct a.seat_id
FROM            cinema a
JOIN            cinema b
ON              ABS(a.seat_id - b.seat_id) = 1
                AND a.free = true and b.free = true
ORDER BY        a.seat_id;
```

```
Cinema table:
+---------+------+
| seat_id | free |
+---------+------+
| 1       | 1    |
| 2       | 0    |
| 3       | 1    |
| 4       | 1    |
| 5       | 1    |
+---------+------+
```

```
Output:
+---------+
| seat_id |
+---------+
| 3       |
| 4       |
| 5       |
+---------+
```

SQL

Exercise

610. Triangle Judgement

```
Table: Triangle

+-------------+------+
| Column Name | Type |
+-------------+------+
| x           | int  |
| y           | int  |
| z           | int  |
+-------------+------+
(x,   y,   z)   is   the
primary key column for
this table.
Each row of this table
contains   the   lengths
of       three       line
segments.
```

```
Triangle table:
+----+----+----+
| x  | y  | z  |
+----+----+----+
| 13 | 15 | 30 |
| 10 | 20 | 15 |
+----+----+----+
```

Write an SQL query to report for every three line segments whether they can form a triangle.

Return the result table in any order.

```
Output:
+----+----+----+----------+
| x  | y  | z  | triangle |
+----+----+----+----------+
| 13 | 15 | 30 | No       |
| 10 | 20 | 15 | Yes      |
+----+----+----+----------+
```

SQL

Answer

```
Triangle table:
+----+----+----+
| x  | y  | z  |
+----+----+----+
| 13 | 15 | 30 |
| 10 | 20 | 15 |
+----+----+----+
```

```
Output:
+----+----+----+----------+
| x  | y  | z  | triangle |
+----+----+----+----------+
| 13 | 15 | 30 | No       |
| 10 | 20 | 15 | Yes      |
+----+----+----+----------+
```

```
SELECT         x, y, z,
               CASE
                      WHEN x + y > z AND x + z > y AND y + z > x
                              THEN 'Yes'
                      ELSE 'No'
               END AS 'triangle'
FROM           triangle;
```

SQL

Exercise

627. Swap Salary

```
Table: Salary

+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| id          | int      |
| name        | varchar  |
| sex         | ENUM     |
| salary      | int      |
+-------------+----------+
```

id is the primary key for this table.
The sex column is ENUM value of type ('m', 'f').
The table contains information about an employee.

Write an SQL query to swap all 'f' and 'm' values (i.e., change all 'f' values to 'm' and vice versa) with a single update statement and no intermediate temporary tables.

Note that you must write a single update statement, do not write any select statement for this problem.

```
Salary table:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | m   | 2500   |
| 2  | B    | f   | 1500   |
| 3  | C    | m   | 5500   |
| 4  | D    | f   | 500    |
+----+------+-----+--------+
```

```
Output:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | f   | 2500   |
| 2  | B    | m   | 1500   |
| 3  | C    | f   | 5500   |
| 4  | D    | m   | 500    |
+----+------+-----+--------+
```

SQL

Answer

```
Salary table:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | m   | 2500   |
| 2  | B    | f   | 1500   |
| 3  | C    | m   | 5500   |
| 4  | D    | f   | 500    |
+----+------+-----+--------+
```

```
Output:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | f   | 2500   |
| 2  | B    | m   | 1500   |
| 3  | C    | f   | 5500   |
| 4  | D    | m   | 500    |
+----+------+-----+--------+
```

```
UPDATE          salary
SET             sex =  CASE    sex
                               WHEN 'm'
                                       THEN 'f'
                               ELSE 'm'
                       END;
```

SQL

Exercise

1075. Project Employees I

Table: Project

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
```

(project_id, employee_id) is the primary key of this table. employee_id is a foreign key to Employee table. Each row of this table indicates that the employee with employee_id is working on the project with project_id.

Table: Employee

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
```

employee_id is the primary key of this table. Each row of this table contains information about one employee.

Write an SQL query that reports the average experience years of all the employees for each project, rounded to 2 digits.

Return the result table in any order.

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

```
Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

```
+-------------+---------------+
| project_id  | average_years |
+-------------+---------------+
| 1           | 2.00          |
| 2           | 2.50          |
+-------------+---------------+
```

SQL

Answer

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

```
Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

```
+-------------+---------------+
| project_id  | average_years |
+-------------+---------------+
| 1           | 2.00          |
| 2           | 2.50          |
+-------------+---------------+
```

```sql
SELECT          p.project_id,
                ROUND(AVG(e.experience_years), 2) AS average_years
FROM            project p
JOIN            employee e
ON              p.employee_id = e.employee_id
GROUP BY        1;
```

SQL

Exercise

1076. Project Employees II

```
Table: Project

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
```

(project_id, employee_id) is the primary key of this table.
employee_id is a foreign key to Employee table.
Each row of this table indicates that the employee with employee_id is working on the project with project_id.

```
Table: Employee

+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
```

employee_id is the primary key of this table.
Each row of this table contains information about one employee.

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

```
Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

Write an SQL query that reports all the projects that have the most employees.

Return the result table in any order.

```
Output:
+-------------+
| project_id  |
+-------------+
| 1           |
+-------------+
```

SQL

Answer

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

```
Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

```
Output:
+-------------+
| project_id  |
+-------------+
| 1           |
+-------------+
```

```
WITH employee_count AS (SELECT project_id, COUNT(employee_id) AS cnt
                        FROM Project
                        GROUP BY project_id)
SELECT         project_id
FROM           employee_count
WHERE          cnt = (SELECT MAX(cnt)
                      FROM employee_count);
```

SQL

Exercise

1082. Sales Analysis I

```
Table: Sales

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| seller_id   | int     |
| product_id  | int     |
| buyer_id    | int     |
| sale_date   | date    |
| quantity    | int     |
| price       | int     |
+-------------+---------+
```

This table has no primary key, it can have repeated rows.
product_id is a foreign key to the Product table.
Each row of this table contains some information about one sale.

```
Table: Product

+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
| unit_price   | int     |
+--------------+---------+
```

product_id is the primary key of this table.
Each row of this table indicates the name and the price of each product.

Write an SQL query that reports the best seller by total sales price, If there is a tie, report them all.

Return the result table in any order.

```
Output:
+-------------+
| seller_id   |
+-------------+
| 1           |
| 3           |
+-------------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

SQL

Answer

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
SELECT      seller_id
FROM        Sales
GROUP BY    seller_id
HAVING      SUM(price) =    (SELECT SUM(price)
                             FROM Sales
                             GROUP BY seller_id
                             ORDER BY 1 DESC
                             LIMIT 1)
```

```
Output:
+-------------+
| seller_id   |
+-------------+
| 1           |
| 3           |
+-------------+
```

SQL

Exercise

1083. Sales Analysis II

```
Table: Sales

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| seller_id   | int     |
| product_id  | int     |
| buyer_id    | int     |
| sale_date   | date    |
| quantity    | int     |
| price       | int     |
+-------------+---------+
```

This table has no primary key, it can have repeated rows.
product_id is a foreign key to the Product table.
Each row of this table contains some information about one sale.

```
Table: Product

+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
| unit_price   | int     |
+--------------+---------+
```

product_id is the primary key of this table.
Each row of this table indicates the name and the price of each product.

Write an SQL query that reports the buyers who have bought S8 but not iPhone. Note that S8 and iPhone are products present in the Product table.

Return the result table in any order.

```
Output:
+-------------+
| buyer_id    |
+-------------+
| 1           |
+-------------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

SQL

Answer

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
SELECT          b.buyer_id
FROM            Product AS a
JOIN            Sales AS b
ON              a.product_id = b.product_id
GROUP BY        b.buyer_id
HAVING          SUM(a.product_name = 'S8') > 0
                AND SUM(a.product_name = 'iPhone') = 0;
```

```
Output:
+-------------+
| buyer_id    |
+-------------+
| 1           |
+-------------+
```

SQL

Exercise

1084. Sales Analysis III

```
Table: Sales

+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| seller_id   | int     |
| product_id  | int     |
| buyer_id    | int     |
| sale_date   | date    |
| quantity    | int     |
| price       | int     |
+-------------+---------+
```

This table has no primary key, it can have repeated rows.
product_id is a foreign key to the Product table.
Each row of this table contains some information about one sale.

```
Table: Product

+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
| unit_price   | int     |
+--------------+---------+
```

product_id is the primary key of this table.
Each row of this table indicates the name and the price of each product.

Write an SQL query that reports the products that were only sold in the spring of 2019. That is, between 2019-01-01 and 2019-03-31 inclusive.

Return the result table in any order.

```
Output:
+-------------+--------------+
| product_id  | product_name |
+-------------+--------------+
| 1           | S8           |
+-------------+--------------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

SQL

Answer

```
Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+
```

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```sql
SELECT       s.product_id, product_name
FROM         Sales s
LEFT JOIN    Product p
ON           s.product_id = p.product_id
GROUP BY     s.product_id
HAVING       MIN(sale_date) >= CAST('2019-01-01' AS DATE)
AND          MAX(sale_date) <= CAST('2019-03-31' AS DATE)
```

```
Output:
+-------------+--------------+
| product_id  | product_name |
+-------------+--------------+
| 1           | S8           |
+-------------+--------------+
```

SQL

Exercise

1113. Reported Posts

```
Table: Actions

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| post_id       | int     |
| action_date   | date    |
| action        | enum    |
| extra         | varchar |
+---------------+---------+
```

There is no primary key for this table, it may have duplicate rows.
The action column is an ENUM type of ('view', 'like', 'reaction', 'comment', 'report', 'share').
The extra column has optional information about the action, such as a reason for the report or a type of reaction.

Write an SQL query that reports the number of posts reported yesterday for each report reason. Assume today is 2019-07-05.

Return the result table in any order.

```
Input:
Actions table:
+---------+---------+-------------+--------+--------+
| user_id | post_id | action_date | action | extra  |
+---------+---------+-------------+--------+--------+
| 1       | 1       | 2019-07-01  | view   | null   |
| 1       | 1       | 2019-07-01  | like   | null   |
| 1       | 1       | 2019-07-01  | share  | null   |
| 2       | 4       | 2019-07-04  | view   | null   |
| 2       | 4       | 2019-07-04  | report | spam   |
| 3       | 4       | 2019-07-04  | view   | null   |
| 3       | 4       | 2019-07-04  | report | spam   |
| 4       | 3       | 2019-07-02  | view   | null   |
| 4       | 3       | 2019-07-02  | report | spam   |
| 5       | 2       | 2019-07-04  | view   | null   |
| 5       | 2       | 2019-07-04  | report | racism |
| 5       | 5       | 2019-07-04  | view   | null   |
| 5       | 5       | 2019-07-04  | report | racism |
+---------+---------+-------------+--------+--------+
```

```
Output:
+---------------+--------------+
| report_reason | report_count |
+---------------+--------------+
| spam          | 1            |
| racism        | 2            |
+---------------+--------------+
```

SQL

Answer

```
Input:
Actions table:
+---------+---------+-------------+--------+--------+
| user_id | post_id | action_date | action | extra  |
+---------+---------+-------------+--------+--------+
| 1       | 1       | 2019-07-01  | view   | null   |
| 1       | 1       | 2019-07-01  | like   | null   |
| 1       | 1       | 2019-07-01  | share  | null   |
| 2       | 4       | 2019-07-04  | view   | null   |
| 2       | 4       | 2019-07-04  | report | spam   |
| 3       | 4       | 2019-07-04  | view   | null   |
| 3       | 4       | 2019-07-04  | report | spam   |
| 4       | 3       | 2019-07-02  | view   | null   |
| 4       | 3       | 2019-07-02  | report | spam   |
| 5       | 2       | 2019-07-04  | view   | null   |
| 5       | 2       | 2019-07-04  | report | racism |
| 5       | 5       | 2019-07-04  | view   | null   |
| 5       | 5       | 2019-07-04  | report | racism |
+---------+---------+-------------+--------+--------+
```

```
Output:
+---------------+--------------+
| report_reason | report_count |
+---------------+--------------+
| spam          | 1            |
| racism        | 2            |
+---------------+--------------+
```

```
SELECT       extra AS report_reason,count(distinct post_id) AS report_count
FROM         actions
WHERE        action_date = '2019-07-04’ AND action = 'report’
GROUP BY     extra
```

SQL

Exercise

1141. User Activity for the Past 30 Days I

```
Table: Activity

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
```

There is no primary key for this table, it may have duplicate rows.
The activity_type column is an ENUM of type ('open_session', 'end_session', 'scroll_down', 'send_message').
The table shows the user activities for a social media website.
Note that each session belongs to exactly one user.

Write an SQL query to find the daily active user count for a period of 30 days ending 2019-07-27 inclusively. A user was active on someday if they made at least one activity on that day.

Return the result table in any order.

```
Input:
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+
```

```
Output:
+------------+--------------+
| day        | active_users |
+------------+--------------+
| 2019-07-20 | 2            |
| 2019-07-21 | 2            |
+------------+--------------+
```

SQL

Answer

```
Input:
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+
```

```
Output:
+------------+--------------+
| day        | active_users |
+------------+--------------+
| 2019-07-20 | 2            |
| 2019-07-21 | 2            |
+------------+--------------+
```

```
SELECT      activity_date AS day, count(distinct user_id) AS active_users
FROM        activity
WHERE       DATEDIFF('2019-07-27', activity_date) <30
GROUP BY    activity_date
```

SQL

Exercise

1142. User Activity for the Past 30 Days II

```
Table: Activity

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
There is no primary key for this table, it may
have duplicate rows.
The activity_type column is an ENUM of type
('open_session', 'end_session', 'scroll_down',
'send_message').
The table shows the user activities for a
social media website.
Note that each session belongs to exactly one
user.
```

Write an SQL query to find the average number of sessions per user for a period of 30 days ending 2019-07-27 inclusively, rounded to 2 decimal places. The sessions we want to count for a user are those with at least one activity in that time period.

```
Input:
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+
```

```
Output:
+---------------------------+
| average_sessions_per_user |
+---------------------------+
| 1.33                      |
+---------------------------+
```

SQL

Answer

```
Input:
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+
```

```
Output:
+---------------------------+
| average_sessions_per_user |
+---------------------------+
| 1.33                      |
+---------------------------+
```

```
SELECT    IFNULL(ROUND(COUNT(DISTINCT session_id)/COUNT(DISTINCT user_id), 2),0.00)
          AS average_sessions_per_user
FROM      Activity
WHERE     activity_date BETWEEN '2019-06-28' AND '2019-07-27'
```

SQL

Exercise

1173. Immediate Food Delivery I

```
Table: Delivery

+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
delivery_id is the primary key of this table.
```

The table holds information about food delivery to customers that make orders at some date and specify a preferred delivery date (on the same order date or after it).

```
Input:
Delivery table:
+-------------+-------------+------------+-----------------------------+
| delivery_id | customer_id | order_date | customer_pref_delivery_date |
+-------------+-------------+------------+-----------------------------+
| 1           | 1           | 2019-08-01 | 2019-08-02                  |
| 2           | 5           | 2019-08-02 | 2019-08-02                  |
| 3           | 1           | 2019-08-11 | 2019-08-11                  |
| 4           | 3           | 2019-08-24 | 2019-08-26                  |
| 5           | 4           | 2019-08-21 | 2019-08-22                  |
| 6           | 2           | 2019-08-11 | 2019-08-13                  |
+-------------+-------------+------------+-----------------------------+
```

If the customer's preferred delivery date is the same as the order date, then the order is called immediate; otherwise, it is called scheduled.

Write an SQL query to find the percentage of immediate orders in the table, rounded to 2 decimal places.

```
Output:
+----------------------+
| immediate_percentage |
+----------------------+
| 33.33                |
+----------------------+
```

SQL

Answer

```
Input:
Delivery table:
+-------------+-------------+------------+-----------------------------+
| delivery_id | customer_id | order_date | customer_pref_delivery_date |
+-------------+-------------+------------+-----------------------------+
| 1           | 1           | 2019-08-01 | 2019-08-02                  |
| 2           | 5           | 2019-08-02 | 2019-08-02                  |
| 3           | 1           | 2019-08-11 | 2019-08-11                  |
| 4           | 3           | 2019-08-24 | 2019-08-26                  |
| 5           | 4           | 2019-08-21 | 2019-08-22                  |
| 6           | 2           | 2019-08-11 | 2019-08-13                  |
+-------------+-------------+------------+-----------------------------+
```

```
Output:
+----------------------+
| immediate_percentage |
+----------------------+
| 33.33                |
+----------------------+
```

```
SELECT          ROUND(100*AVG(order_date = customer_pref_delivery_date), 2)
                AS immediate_percentage
FROM            Delivery;
```

SQL

Exercise

176. Second Highest Salary

```
Table: Employee

+-------------+------+
| Column Name | Type |
+-------------+------+
| id          | int  |
| salary      | int  |
+-------------+------+
id is the primary key column for this
table.
Each row of this table contains
information about the salary of an
employee.
```

```
Input:
Employee table:
+----+--------+
| id | salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

Write an SQL query to report the second highest salary from the Employee table. If there is no second highest salary, the query should report null.

```
Output:
+---------------------+
| SecondHighestSalary |
+---------------------+
| 200                 |
+---------------------+
```

SQL

Answer

```
Input:
Employee table:
+----+--------+
| id | salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

```
Output:
+---------------------+
| SecondHighestSalary |
+---------------------+
| 200                 |
+---------------------+
```

```
SELECT
                (SELECT        DISTINCT Salary
                 FROM           Employee
                 ORDER BY Salary DESC
                 LIMIT 1 OFFSET 1) AS SecondHighestSalary
```

SQL

Exercise

612. Shortest Distance in a Plane

```
Table: Point2D

+-------------+------+
| Column Name | Type |
+-------------+------+
| x           | int  |
| y           | int  |
+-------------+------+
(x, y) is the primary key column for
this table.
Each row of this table indicates the
position of a point on the X-Y plane.
```

```
Input:
Point2D table:
+----+----+
| x  | y  |
+----+----+
| -1 | -1 |
| 0  | 0  |
| -1 | -2 |
+----+----+
```

The distance between two points p1(x1, y1) and p2(x2, y2) is sqrt((x2 - x1)2 + (y2 - y1)2).

Write an SQL query to report the shortest distance between any two points from the Point2D table. Round the distance to two decimal points.

```
Output:
+----------+
| shortest |
+----------+
| 1.00     |
+----------+
```

SQL

Answer

```
Input:
Point2D table:
+----+----+
| x  | y  |
+----+----+
| -1 | -1 |
| 0  | 0  |
| -1 | -2 |
+----+----+
```

```
Output:
+----------+
| shortest |
+----------+
| 1.00     |
+----------+
```

```
SELECT      p1.x, p1.y, p2.x, p2.y,
            SQRT((POW(p1.x - p2.x, 2) + POW(p1.y - p2.y, 2))) AS distance
FROM        point_2d p1
JOIN        point_2d p2
ON          p1.x != p2.x OR p1.y != p2.y
```

```
SELECT      ROUND(SQRT(MIN((POW(p1.x - p2.x, 2) + POW(p1.y - p2.y, 2)))), 2)
            AS shortest
FROM        point_2d p1
JOIN        point_2d p2
ON          p1.x != p2.x OR p1.y != p2.y
```

SQL

Exercise

1107. New Users Daily Count

```
Table: Traffic

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| activity      | enum    |
| activity_date | date    |
+---------------+---------+
There is no primary key for this table,
it may have duplicate rows.
The activity column is an ENUM type of
('login', 'logout', 'jobs', 'groups',
'homepage').
```

Write an SQL query to reports for every date within at most 90 days from today, the number of users that logged in for the first time on that date. Assume today is 2019-06-30.

Return the result table in any order.

```
Input:
Traffic table:
+---------+----------+---------------+
| user_id | activity | activity_date |
+---------+----------+---------------+
| 1       | login    | 2019-05-01    |
| 1       | homepage | 2019-05-01    |
| 1       | logout   | 2019-05-01    |
| 2       | login    | 2019-06-21    |
| 2       | logout   | 2019-06-21    |
| 3       | login    | 2019-01-01    |
| 3       | jobs     | 2019-01-01    |
| 3       | logout   | 2019-01-01    |
| 4       | login    | 2019-06-21    |
| 4       | groups   | 2019-06-21    |
| 4       | logout   | 2019-06-21    |
| 5       | login    | 2019-03-01    |
| 5       | logout   | 2019-03-01    |
| 5       | login    | 2019-06-21    |
| 5       | logout   | 2019-06-21    |
+---------+----------+---------------+
```

```
Output:
+------------+-------------+
| login_date | user_count  |
+------------+-------------+
| 2019-05-01 | 1           |
| 2019-06-21 | 2           |
+------------+-------------+
```

SQL

Answer

```
SELECT   login_date, count(user_id) AS user_count
FROM     (SELECT user_id, min(activity_date) AS login_date
          FROM Traffic
          WHERE activity = 'login’
          GROUP BY user_id) t
WHERE    datediff('2019-06-30', login_date) <= 90
GROUP BY login_date
```

```
SELECT   login_date, count(user_id) AS user_count
FROM     (SELECT user_id, min(activity_date) AS login_date
          FROM Traffic
          WHERE activity = 'login’
          GROUP BY user_id) t
WHERE    login_date between date_add('2019-06-30',
interval -90 day) and '2019-06-30'
GROUP BY login_date
```

```
Input:
Traffic table:
+---------+----------+---------------+
| user_id | activity | activity_date |
+---------+----------+---------------+
| 1       | login    | 2019-05-01    |
| 1       | homepage | 2019-05-01    |
| 1       | logout   | 2019-05-01    |
| 2       | login    | 2019-06-21    |
| 2       | logout   | 2019-06-21    |
| 3       | login    | 2019-01-01    |
| 3       | jobs     | 2019-01-01    |
| 3       | logout   | 2019-01-01    |
| 4       | login    | 2019-06-21    |
| 4       | groups   | 2019-06-21    |
| 4       | logout   | 2019-06-21    |
| 5       | login    | 2019-03-01    |
| 5       | logout   | 2019-03-01    |
| 5       | login    | 2019-06-21    |
| 5       | logout   | 2019-06-21    |
+---------+----------+---------------+
```

```
Output:
+------------+-------------+
| login_date | user_count  |
+------------+-------------+
| 2019-05-01 | 1           |
| 2019-06-21 | 2           |
+------------+-------------+
```

SQL

Exercise

1112. Highest Grade For Each Student

```
Table: Enrollments

+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| course_id     | int     |
| grade         | int     |
+---------------+---------+
(student_id, course_id) is the primary
key of this table.
```

```
Input:
Enrollments table:
+------------+-------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 2          | 2         | 95    |
| 2          | 3         | 95    |
| 1          | 1         | 90    |
| 1          | 2         | 99    |
| 3          | 1         | 80    |
| 3          | 2         | 75    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

Write a SQL query to find the highest grade with its corresponding course for each student. In case of a tie, you should find the course with the smallest course_id.

Return the result table ordered by student_id in ascending order.

```
Output:
+------------+-------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 1          | 2         | 99    |
| 2          | 2         | 95    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

SQL

Answer

```
Input:
Enrollments table:
+------------+--------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 2          | 2         | 95    |
| 2          | 3         | 95    |
| 1          | 1         | 90    |
| 1          | 2         | 99    |
| 3          | 1         | 80    |
| 3          | 2         | 75    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

```
Output:
+------------+--------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 1          | 2         | 99    |
| 2          | 2         | 95    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

```
SELECT         student_id, MIN(course_id) AS "course_id", grade AS "grade"
FROM           Enrollments
WHERE          (student_id, grade) IN
                      (SELECT       student_id, max(grade)
                       FROM         Enrollments
                       GROUP BY     student_id)
GROUP BY       student_id
ORDER BY       student_id
```