Learning Objectives

Learners will be able to...

- explain table structure by schema
- use aggregate functions
- use aliases
- use WHERE predicate
- sort query results
- group results
- filter grouped results

info

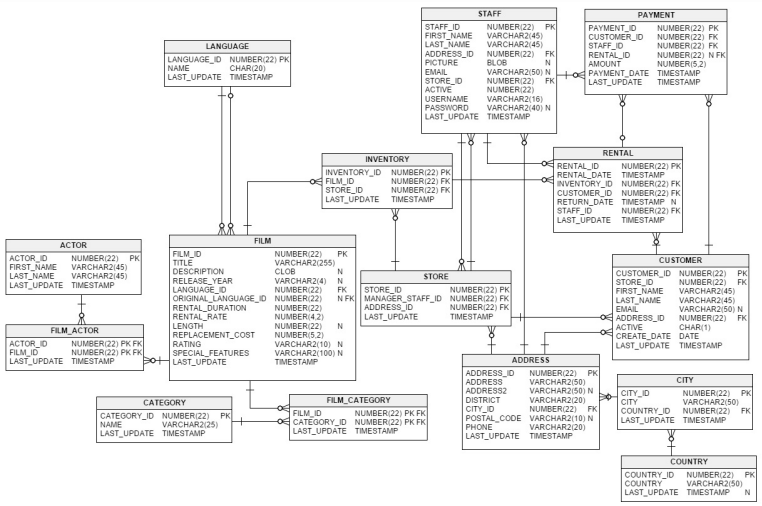
Make Sure You Know

A primary key uniquely identifies each record in a table, ensuring data integrity. A foreign key establishes a relationship between tables by referencing the primary key in another table.

Reading database description

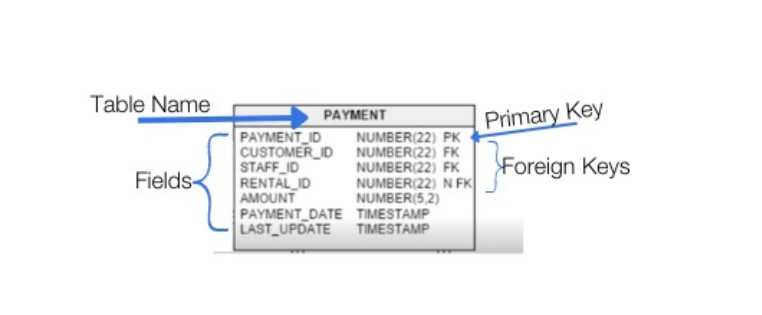
The ER-diagram (entity-relationship diagram) will help you understand the structure of the database. An ER diagram graphically displays the relationships of tables in a database.

We are using The Sakila Database and the full database structure will looks like this:



The Sakila Database has multiple tables and shows the relations of the tables with the ERD Cardinality

But for now, we don't need to go that deep and we'll look at the structure of just one table - `payment`.



One table with defined table name and it shows the primary and foreign keys

The label PK (primary key) marks the field that is the primary key in the table.
The FK (foreign key) label marks a field that is present in other tables and is a key in them.
We also see the display of the field names and their types.

Knowing the name of the fields in the table and their types, we can make various queries to retrieve data from it.

By paste `\d payment` in psql console you could retrieve actual database structure. It will look like:

```
                                          Table
"public.payment"
    Column    |            Type             | Collation |
Nullable |                  Default
--------------+-----------------------------+-----------+-------
---+-------------------------------------------
 payment_id   | integer                     |           | not
null | nextval('payment_payment_id_seq'::regclass)
 customer_id  | integer                     |           | not
null |
 staff_id     | integer                     |           | not
null |
 rental_id    | integer                     |           | not
null |
 amount       | numeric(5,2)                |           | not
null |
 payment_date | timestamp without time zone |           | not
null |
Indexes:
    "payment_pkey" PRIMARY KEY, btree (payment_id)
    "idx_fk_customer_id" btree (customer_id)
    "idx_fk_staff_id" btree (staff_id)
Foreign-key constraints:
    "payment_customer_id_fkey" FOREIGN KEY (customer_id)
REFERENCES customer(customer_id) ON UPDATE CASCADE ON DELETE
RESTRICT
    "payment_rental_id_fkey" FOREIGN KEY (rental_id) REFERENCES
rental(rental_id) ON UPDATE CASCADE ON DELETE SET NULL
    "payment_staff_id_fkey" FOREIGN KEY (staff_id) REFERENCES
staff(staff_id) ON UPDATE CASCADE ON DELETE RESTRICT
```

Aggregate Functions

SQL has functions for counting the total number of rows, sum, average, maximum and minimum. Such functions are called aggregate functions.

Function	Return Type	Description
AVG(expression)	numeric for any integer-type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all non-null input values
COUNT(*)	bigint	number of input rows
COUNT(expression)	bigint	number of input rows for which the value of expression is not null
MAX(expression)	same as argument type	maximum value of expression across all non-null input values
MIN(expression)	same as argument type	minimum value of expression across all non-null input values
SUM(expression)	bigint for smallint or int arguments, numeric for bigint arguments, otherwise the same as the argument data type	sum of expression across all non-null input values

So we can calculate the maximum, minimum and amount of payments in the `payment` table.

```
SELECT MIN(amount), MAX(amount), SUM(amount) FROM payment;
```

Try to execute query and check the result.

We can also calculate data for a specific user by adding the condition WHERE:

```
SELECT MIN(amount), MAX(amount), SUM(amount)
FROM payment WHERE customer_id = 148;
```

info

Difference between COUNT(*) and COUNT(expression)

These expressions will produce the same value (number of rows in the table) as long as there are no empty(`NULL`) values in the rows that fall under the expression.

In addition to built-in functions, you can perform mathematical operations on data, like `+`, `-`, `/`, `*`, etc.

```
SELECT payment_id + payment_id  FROM payment WHERE customer_id =
        148;
```

Or even combine them.

```
SELECT MIN(amount) + MAX(amount) FROM payment WHERE customer_id
        = 148;
```

AS Statement

During execution query below you could notice strange column name for expression: `?column?`. You may see this for compound expressions for which the DBMS cannot use the column name.

```
SELECT MIN(amount) + MAX(amount) FROM payment WHERE customer_id
= 148;

 ?column?
----------
    11.98
(1 row)
```

But with the help of the **AS** operator, you can set the name yourself.

```
SELECT MIN(amount) + MAX(amount) AS min_plus_max
FROM payment WHERE customer_id = 148;
```

The name of column will be `min_plus_max`.

In addition to expressions SELECT, we can also set aliases for table names. This will be especially convenient when we have to work with joining several tables that have long names.

```
SELECT MIN(p.amount) + MAX(p.amount) AS min_plus_max
FROM payment AS p WHERE p.customer_id = 148;
```

In the example above, we gave the `payment` table a short name `p` and used it to access the fields, like `p.customer_id`.

WHERE Statement

Where picture

The **WHERE** operator is used to select specific values.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

You have seen the example before:

```
SELECT MIN(amount), MAX(amount), SUM(amount)
FROM payment WHERE customer_id = 148;
```

But this was the simplest example, in this expression you can compose complex conditions using various logical operators.

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal
—	—
AND	Logical operator AND
OR	Logical operator OR
IN	Return true if a value matches any value in a list

BETWEEN	Return true if a value is between a range of values
LIKE	Return true if a value matches a pattern
IS NULL	Return true if a value is NULL
NOT	Negate the result of other operators

Query below will select records with payment amount more then 10, payment date above 2005-08-23 processed by staff with id 1.

```
SELECT * FROM payment
WHERE amount > 10 AND payment_date > '2005-08-23' AND staff_id =
        1;
```

LIKE is a logical operator that is used to retrieve the data in a column of a table, based on a specified pattern.

WildCard	Definition
%	The percent sign represents zero, one or multiple characters.
_	The underscore represents a single number or character.
[]	This matches any single character within the given range in the [].
[^]	This matches any single character excluding the given range in the [^]

Query will select all actors with name started with J. Operator is case sensitive.

```
SELECT * FROM actor WHERE first_name LIKE 'J%';
```

info

NULL values comparisons

Comparison operators like =, <, or <> cannot be used to check for NULL values. Instead, we use the following operators.

- **IS NULL**
- **IS NOT NULL** (negation of NULL values)

GROUP BY Statement

Previously, we were able to calculate the minimum, maximum and amount of sales for a particular user. But what if we want to get slices for each of the users in the form of a pivot table?

There is a grouping operator for this purpose - **GROUP BY**.

The `GROUP BY` statement is used to group rows based on one or more columns and apply aggregate functions to each group. It allows us to perform calculations and analysis on subsets of data within a table.

When using the `GROUP BY` statement, we specify the column(s) we want to group by in the `SELECT` statement. The result will be a set of rows where each row represents a unique combination of the grouped columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s);
```

This way we can get grouped values by users:

```
SELECT MIN(amount), MAX(amount), SUM(amount), customer_id
FROM payment GROUP BY customer_id LIMIT 10;
```

In the example provided, we are grouping the payment data by the `customer_id` column. This means that the calculation of `MIN`, `MAX`, and `SUM` will be performed for each unique `customer_id`.

ORDER BY Clause

A feature of SQL is that the data from the additional directives can be returned in an arbitrary order. We are usually used to dealing with rows arranged in a certain order.

For example, in the previous query, we would like to know the top maximum buyers. And we can do it with sort operator: **ORDER BY**.

```
 min  |  max  |  sum   | customer_id
------+-------+--------+-------------
 0.99 |  9.99 | 118.68 |           1
 0.99 | 10.99 | 128.73 |           2
 0.99 | 10.99 | 135.74 |           3
 0.99 |  8.99 |  81.78 |           4
 0.99 |  9.99 | 144.62 |           5
 0.99 |  7.99 |  93.72 |           6
 0.99 |  8.99 | 151.67 |           7
 0.99 |  9.99 |  92.76 |           8
 0.99 |  7.99 |  89.77 |           9
 0.99 |  8.99 |  99.75 |          10
(10 rows)
```

The `ORDER BY` keyword is used to sort the result-set in ascending or descending order.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

```
SELECT MIN(amount), MAX(amount), SUM(amount), customer_id
FROM payment GROUP BY customer_id
ORDER BY SUM(amount) LIMIT 10;
```

We got the order from smallest to largest, but this is not quite what we planned.

The `ORDER BY` keyword sorts the records in ascending order by default. To sort the records in descending order, use the `DESC` keyword. We can also set an alias and use it for sorting.

```
SELECT MIN(amount), MAX(amount), SUM(amount) AS s, customer_id
FROM payment GROUP BY customer_id
ORDER BY s DESC LIMIT 10;
```

Now the sort order is what we needed.

We can list fields separated by commas for additional sorting.

```
SELECT MIN(amount), MAX(amount) AS m, SUM(amount) AS s,
       customer_id
FROM payment GROUP BY customer_id
ORDER BY s DESC, m LIMIT 10;
```

Also, as an argument of this operator, you can use a number - the ordinal number of the column for sorting.

```
SELECT MIN(amount), MAX(amount), SUM(amount), customer_id
FROM payment GROUP BY customer_id ORDER BY 1 DESC LIMIT 10;
```

HAVING Clause

The HAVING clause was introduced in SQL to enable filtering based on conditions involving aggregate functions, which cannot be achieved using the WHERE keyword alone. It allows us to apply conditions to grouped data after aggregations have been performed.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition;
```

For example, we can get users whose payment amount is more than 200 in the following way.

```
SELECT MIN(amount), MAX(amount) AS m, SUM(amount) AS s,
        customer_id
FROM payment GROUP BY customer_id
HAVING SUM(amount) > 200
ORDER BY s DESC, m LIMIT 10;
```

info

Note that you cannot use the **WHERE** operator for the results of aggregated functions, this will throw an error.