DBMS Lab 3 Demo

All examples here are with respect to the dvdrental database provided to you.

Aggregation functions

Aggregate functions perform a calculation on a set of rows and return a single row.
We shall be looking into 5 Aggregation functions:

Count

The `COUNT()` function is an aggregate function that allows you to obtain the number of rows that match a specific condition.
The `COUNT(*)` function returns the number of rows returned by a `SELECT` statement, including NULL and duplicates.
The `COUNT(column_name)` function returns the number of rows returned by a `SELECT` clause. However, it does not consider `NULL` values in the `column_name`.
The `COUNT(DISTINCT column_name)` returns the number of unique non-null values in the `column_name`.

syntax:

```
SELECT COUNT([ * | column_name | distinct column name ])
FROM table_name WHERE condition;
```

Example:
Query:

```
select count(title) from film where rating = 'G';
```

Output:

	count bigint
1	178

Max

The `MAX()` function is an aggregate function that returns the maximum value in a set of values. You can use the `MAX()` function not just in the `SELECT` clause but also in the `WHERE` and `HAVING` clauses.

syntax:

```
SELECT MAX(expression) FROM table_name;
```

Example:
Query:

```
select max(payment_date) from payment;
```

Output:

	max timestamp without time zone
1	2007-05-14 13:44:29.996577

Min

The `MIN()` function is an aggregate function that returns the minimum value in a set of values. The `DISTINCT` option does not have any effects on the `MIN()` function.

syntax:

```
SELECT MIN (expression) FROM table_name;
```

Avg

The `AVG()` function allows you to calculate the average value of a set.
To calculate the average value of distinct values in a set, you use the distinct option as follows:
Notice that the `AVG()` function ignores `NULL`. If the column has no values, the `AVG()` function returns `NULL`.

syntax:

```
SELECT AVG([column_name | distinct column_name]) FROM table_name;
```

Sum

The `SUM()` is an aggregate function that returns the sum of values in a set.
The `SUM()` function ignores `NULL`, meaning that it doesn't consider the `NULL` in calculation.
If you use the `DISTINCT` option, the `SUM()` function calculates the sum of only distinct values.
The `SUM()` of an empty set will return `NULL`, not zero.

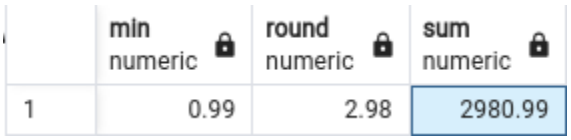
syntax:

```
SELECT SUM([ column_name | distinct column_name]) FROM payment;
```

Example:
Query:

```
select min(rental_rate), round(avg(rental_rate), 2), sum(rental_rate) from
film;
```

Output:



	min numeric	round numeric	sum numeric
1	0.99	2.98	2980.99

Grouping

Group By

The GROUP BY statement is used to group different rows based on the values of the specified columns. Here, we have to use aggregation functions mentioned above on each group.

Some syntatic rules of group by:

- In the query, the GROUP BY clause is placed after the WHERE clause.
- In the query, the GROUP BY clause is placed before the ORDER clause if used.
- In the query, the GROUP BY clause is placed before the HAVING clause.

syntax:

```
SELECT column1, aggregate_function(column2) FROM table_name GROUP BY
column1;
```

Example:
Query:

```
select title, count(actor_id) as no_actors from film
join film_actor on film.film_id = film_actor.film_id
group by title order by title limit 5;
```

Output:

	title character varying (255)	no_actors bigint
1	Academy Dinosaur	10
2	Ace Goldfinger	4
3	Adaptation Holes	5
4	Affair Prejudice	5
5	African Egg	5

Having

HAVING statement is used to place conditions on the groups created by the GROUP BY statement.

syntax:

```
SELECT column1, AGGREGATE_FUNCTION(column2) FROM table1 GROUP BY column1
HAVING condition;
```

Example:
Query:

```
select title, length, count(actor_id) as no_actors from film
join film_actor on film.film_id = film_actor.film_id
group by title,length having length < 60 order by title limit 5;
```

Output:

	title character varying (255)	length smallint	no_actors bigint
1	Ace Goldfinger	48	4
2	Adaptation Holes	50	5
3	Airport Pollock	54	4
4	Alien Center	46	6
5	Alter Victory	57	4

Nested queries

A nested query (also called a subquery) is a query embedded within another SQL query. The result of the inner query is used by the outer query to perform further operations. Nested queries are commonly used for filtering data, performing calculations, or joining datasets indirectly.

The inner query runs before the outer query.

Types of Nested Queries:

Independent Queries:

The execution of the inner query is not dependent on the outer query. The result of the inner query is used directly by the outer query.

Example:
Query:

```
select count(*) from (
select count(rental_date) from rental
where rental_date BETWEEN '28-06-2005' and '12-07-2005'
group by customer_id having count(rental_date) > 1
) as rent;
```

Output:

	count bigint
1	577

Correlated Queries:

The inner query depends on the outer query for its execution. For each row processed by the outer query, the inner query is executed. The EXISTS keyword is often used with correlated queries.

Example:
Query:

```
select title from film f where exists (
      select 1 from film_actor fa where f.film_id = fa.film_id and
      fa.actor_id in (select actor_id from actor where first_name like
'A%')
) limit 5;
```

Output:

	title character varying (255)
1	Aladdin Calendar
2	Alaska Phantom
3	Alley Evolution
4	Alter Victory
5	Annie Identity

Some Operators used with Nested Queries:

IN:
The `IN` operator is used to filter rows based on a set of values returned by a subquery. It is commonly used to match a column value against a list or result set. This operator simplifies queries by avoiding the need for multiple `OR` conditions.

syntax:

```
SELECT [attributes] FROM table_name
WHERE attribute IN [ view | table | list of attributes ];
```

NOT IN:
The `NOT IN` operator excludes rows based on a set of values from a subquery. It is particularly useful for filtering out unwanted results. This operator helps identify records that do not match the conditions defined in the subquery.

syntax:

```
SELECT [attributes] FROM table_name
WHERE attribute IN [ view | table | list of attributes ];
```

EXISTS:
The `EXISTS` operator checks for the existence of rows in a subquery. It returns true if the subquery produces any rows, making it efficient for conditional checks. This operator is often used to test for relationships between tables.

syntax:

```
SELECT [attributes] FROM table_name WHERE EXISTS sub_query;
```

ANY
The `ANY` operator compares a value with any value returned by the subquery

syntax:

```
SELECT [attributes] FROM table_name WHERE attribute > ANY(suq_query);
```

ALL
ALL ensures comparison with all values.

syntax:

```
SELECT [attributes] FROM table_name WHERE attribute > ALL(suq_query);
```

Best Practices

1. **Prefer joins for simple queries:** For cases where nested queries add unnecessary complexity, consider using joins instead. Joins are typically faster and more readable for straightforward relationships.
2. **Avoid deep nesting:** Limit the levels of nesting in queries to improve performance and maintain readability. Deeply nested queries can be computationally expensive and difficult to debug.
3. **Use** `EXISTS` **and** `IN` **wisely:** Choose the appropriate operator based on the scenario. Use `EXISTS` for checking existence and `IN` for comparing with a list of values.

Casing

The `CASE` expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the `ELSE` clause.
If there is no `ELSE` part and no conditions are true, it returns NULL.

syntax:

```
CASE
      WHEN condition1 THEN result1
       WHEN condition2 THEN result2
      WHEN conditionN THEN resultN
      ELSE result
END;
```

Example:
Query:

```
select customer.first_name||' '||customer.last_name as Fullname,
sum(amount), case
```

```
when sum(amount)<=50 then 'Bronze'
when sum(amount)>50 and sum(amount)<=100 then 'Silver'
when sum(amount)>100 and sum(amount)<=150 then 'Gold'
else 'Platinum'
end as category
from customer,payment
where payment.customer_id = customer.customer_id
group by customer.customer_id order by category limit 5;
```

Output:

	fullname text	sum numeric	category text
1	Leona Obrien	32.90	Bronze
2	Brian Wyman	27.93	Bronze
3	Tiffany Jordan	49.88	Bronze
4	Anthony Schwab	47.85	Bronze
5	Caroline Bowman	37.87	Bronze

Processing Timestamps

We shall be focusing on timestamp data without any time zone values.
The `timestamp` data type allows you to store both date and time. However, it does not have any time zone data. It means that when you change the timezone of your database server, the timestamp value stored in the database will not change automatically.

Format:

```
'YYYY-MM-DD HH:MM:SS-MS'
```

Example data: '2016-06-22 19:10:25-07'

To get current time:

```
SELECT CURRENT_TIMESTAMP;
SELECT NOW();
```

To get current time without date:

```
SELECT CURRENT_TIME;
```

To get the time of day in the string format, you use the `timeofday()` function.

```
SELECT TIMEOFDAY();
```

To define a default value for the timestamp attributes:

```
CREATE TABLE table_name (
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
);
```

You can compare different timestamp values like:

```
select count(rental_date) from rental where rental_date BETWEEN
'28-06-2005' and '12-07-2005';
```

You can also use `MIN` and `MAX` aggregation functions on them.

To extract different features from the data, we do the following:

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2025-01-31 13:30:15');
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2025-01-31 13:30:15');
SELECT EXTRACT(MONTH FROM TIMESTAMP '2025-01-31 13:30:15');
```

To extract the total time in seconds from an interval:

```
SELECT EXTRACT(EPOCH FROM (SELECT (NOW() - '2025-01-01 00:00:00')));
```

Type Casting

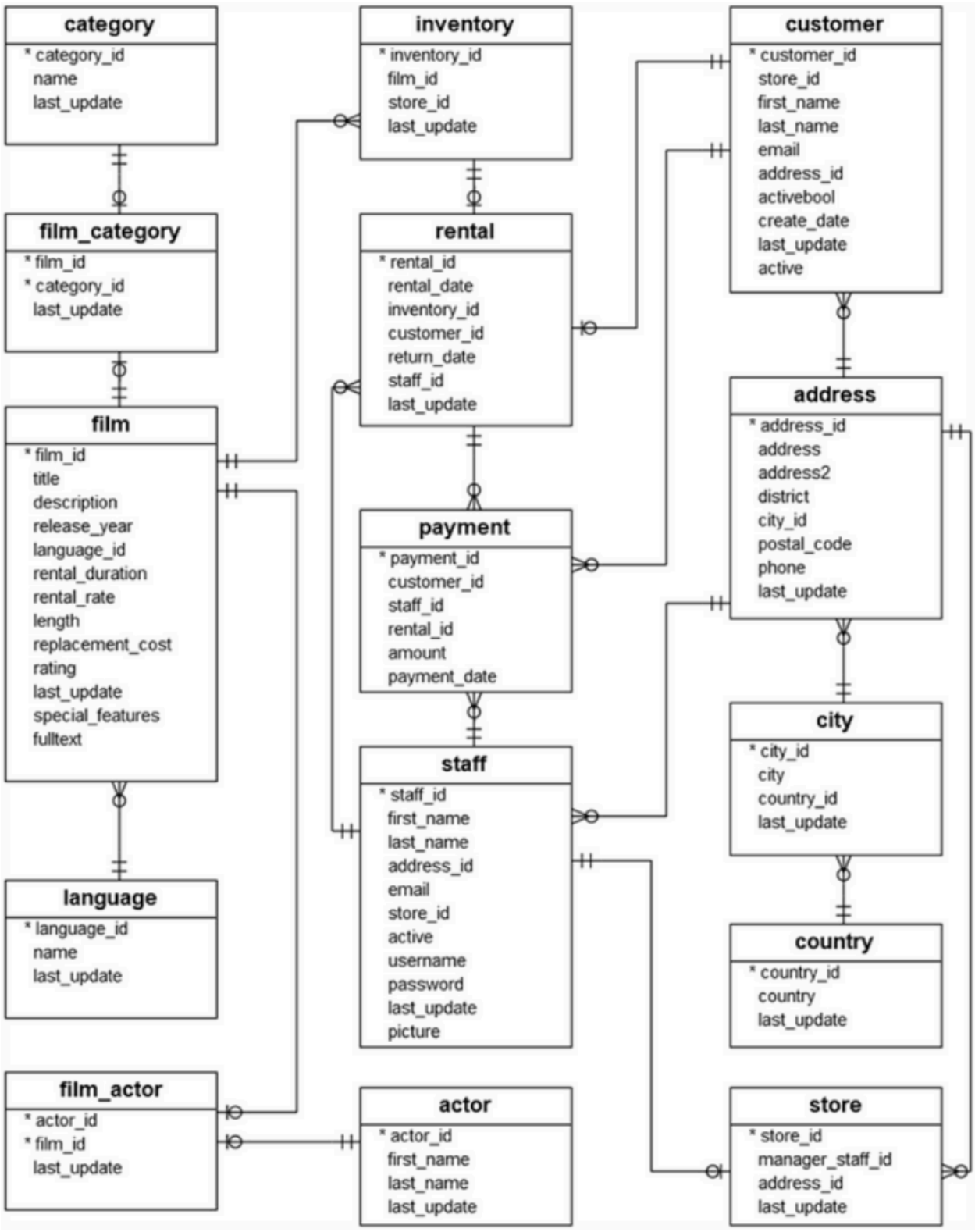
You can explicitly type cast data using CAST command
Another method is using the :: operator

syntax:

```
CAST ( expression AS target_type );
SELECT expression :: target_type;
```

example:
query:

```
select cast('100' as integer);
select 100::integer;
```


category
* category_id
name
last_update
film_category
* film_id
* category_id
last_update
film
* film_id
title
description
release_year
language_id
rental_duration
rental_rate
length
replacement_cost
rating
last_update
special_features
fulltext
language
* language_id
name
last_update
film_actor
* actor_id
* film_id
last_update
inventory
* inventory_id
film_id
store_id
last_update
rental
* rental_id
rental_date
inventory_id
customer_id
return_date
staff_id
last_update
payment
* payment_id
customer_id
staff_id
rental_id
amount
payment_date
staff
* staff_id
first_name
last_name
address_id
email
store_id
active
username
password
last_update
picture
actor
* actor_id
first_name
last_name
last_update
customer
* customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active
address
* address_id
address
address2
district
city_id
postal_code
phone
last_update
city
* city_id
city
country_id
last_update
country
* country_id
country
last_update
store
* store_id
manager_staff_id
address_id
last_update