MySQL 과 윈도우 함수

- 윈도우 함수는 SQL 쿼리 내에서 데이터 집합을 세분화하여 각 부분에 대한 계산을 수행하는 함수임
- 이를 통해 데이터의 순위를 매기거나, 집계, 이동 평균, 누적 합 등을 계산할 수 있으며, 기본 집계 함수보다 더 유연하게 데이터를 분석할 수 있음
- 이 함수들은 특정 '윈도우'(데이터의 부분 집합) 내에서 작동하며, 각각의 행에 대해 결과를 반환하되, 전체 쿼리 결과의 컨텍스트 내에서 실행됨
- **MySQL 8.0 (2018년 4월 출시)** 버전부터 윈도우 함수가 처음으로 지원되기 시작했음

윈도우 함수:

1. **행 기준 연산**: 각 행에 대해 연산을 수행하면서, 원본 행의 구조를 유지합니다. 이는 각 행에 대한 상세 정보를 보존하면서 계산을 수행할 수 있게 해줍니다.
2. **부분 데이터 집합 사용**: 특정 윈도우(데이터의 부분 집합) 내에서 연산을 수행하고, 이는 `PARTITION BY` 절을 통해 더 세분화할 수 있습니다.
3. **다양한 연산 지원**: 순위 매기기, 누적 집계, 이동 평균 등 복잡한 계산을 가능하게 합니다.
4. **원본 데이터셋 변경 없음**: 원본 데이터의 행 수와 구조는 변경되지 않으며, 계산 결과가 각 행에 추가됩니다.
5. **집계와 상세 데이터 동시 제공**: 원본 데이터의 상세 정보와 함께 집계 결과를 제공할 수 있어, 데이터의 맥락을 유지하며 분석할 수 있습니다.

GROUP BY:

1. **그룹 기준 연산**: 데이터를 특정 열의 값에 따라 그룹화하고, 각 그룹에 대한 집계 연산(합계, 평균, 최대값 등)을 수행합니다.
2. **데이터 집약**: `GROUP BY` 는 데이터를 그룹화하여 새로운 집합을 생성하고, 이 집합은 원본 데이터보다 행 수가 적을 수 있습니다.
3. **단순 집계 연산 제한**: 주로 집계 함수를 사용하여 그룹별 합계나 평균 등을 계산합니다.
4. **원본 데이터셋 축소**: 결과 데이터셋은 그룹화된 요약 정보만 포함하며, 각 그룹별로 단 하나의 결과를 반환합니다.
5. **상세 데이터 제공 불가**: 그룹화 과정에서 원본 행의 상세 정보는 손실되며, 각 그룹의 대표 값만을 제공합니다.

RANK(), DENSE_RANK(), ROW_NUMBER() 문법

SQL에서 `RANK()` , `DENSE_RANK()` , `ROW_NUMBER()` 함수는 행의 순위나 순서를 매기는 데 사용되는 윈도우 함수입니다. 이 함수들은 각각 조금씩 다른 방식으로 순위를 매깁니다.

1. `RANK()` : 이 함수는 순위를 매기되, 동일한 값이 있을 경우 같은 순위를 부여하고 다음 순위는 건너뜁니다.

```
RANK() OVER (ORDER BY column_name [ASC|DESC])
```

2. `DENSE_RANK()` : 이 함수도 순위를 매기되, 동일한 값이 있을 경우 같은 순위를 부여하지만 다음 순위는 건너뛰지 않습니다.

```
DENSE_RANK() OVER (ORDER BY column_name [ASC|DESC])
```

3. `ROW_NUMBER()` : 이 함수는 순위와 상관없이 각 행에 고유한 번호를 부여합니다.

```
ROW_NUMBER() OVER (ORDER BY column_name [ASC|DESC])
```

이 함수들은 모두 `OVER` 절과 함께 사용되며, `ORDER BY` 절을 통해 순위를 매길 기준 컬럼을 지정합니다.

sakila 데이터베이스를 이용한 예제

`sakila` 데이터베이스를 사용하여 이 함수들의 사용법을 살펴보겠습니다.

예제 1: 영화 길이에 따른 순위 매기기

다음 쿼리는 `film` 테이블에서 영화 길이(`length`)에 따라 순위를 매깁니다.

```
SELECT
    title,
    length,
    RANK() OVER (ORDER BY length DESC) AS ranking,
    DENSE_RANK() OVER (ORDER BY length DESC) AS dense_ranking,
    ROW_NUMBER() OVER (ORDER BY length DESC) AS row_numbers
FROM
    film
ORDER BY length DESC; -- 최종 결과를 알아보기 쉽게 명시적으로 정렬하기로 함
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| title           | length | rank | dense_rank | row_number |
|-----------------|--------|------|------------|------------|
| Long Movie 1    | 185    | 1    | 1          | 1          |
| Long Movie 2    | 185    | 1    | 1          | 2          |
| Medium Movie 1  | 120    | 3    | 2          | 3          |
| Medium Movie 2  | 120    | 3    | 2          | 4          |
| Short Movie 1   | 90     | 5    | 3          | 5          |
| ...             | ...    | ...  | ...        | ...        |
```

여기서 `RANK()` 와 `DENSE_RANK()` 의 차이를 볼 수 있습니다. `RANK()` 는 같은 길이의 영화에 대해 같은 순위를 매기고 다음 순위를 건너뛰지만, `DENSE_RANK()` 는 같은 순위를 매기더라도 다음 순위는 건너뛰지 않습니다.
`ROW_NUMBER()` 는 각 행에 고유한 번호를 부여합니다.

예제 2: 고객별 총 지불 금액에 따른 순위 매기기

다음 쿼리는 `payment` 테이블에서 고객별 총 지불 금액에 따라 순위를 매깁니다.

```sql
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    SUM(p.amount) AS total_amount,
    RANK() OVER (ORDER BY SUM(p.amount) DESC) AS ranking,
    DENSE_RANK() OVER (ORDER BY SUM(p.amount) DESC) AS dense_ranking,
    ROW_NUMBER() OVER (ORDER BY SUM(p.amount) DESC) AS row_numbers
FROM
    customer c
JOIN
    payment p ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| total_amount | rank | dense_rank | row_number |
|--------------|------|------------|------------|
| 200.00       | 1    | 1          | 1          |
| 200.00       | 1    | 1          | 2          |
| 150.00       | 3    | 2          | 3          |
| ...          | ...  | ...        | ...        |
```

여기서도 `RANK()` 와 `DENSE_RANK()` 의 차이를 볼 수 있습니다. 같은 총 지불 금액을 가진 고객에 대해 `RANK()` 는 같은 순위를 매기고 다음 순위를 건너뛰지만, `DENSE_RANK()` 는 같은 순위를 매기더라도 다음 순위는 건너뛰지 않습니다. `ROW_NUMBER()` 는 각 행에 고유한 번호를 부여합니다.

PARTITION BY, ORDER BY, ROWS/RANGE

SQL 윈도우 함수에서 `PARTITION BY` , `ORDER BY` , `ROWS/RANGE` 는 데이터 집계와 분석을 위해 윈도우(하위 그룹)를 정의하는데 사용됩니다. 이러한 함수들은 데이터를 특정 기준에 따라 세분화하고, 윈도우 내에서 데이터를 정렬 및 범위 지정을 통해 연산을 수행합니다.

1. **PARTITION BY**

- **역할**: `PARTITION BY` 는 특정 컬럼(또는 컬럼 조합)을 기준으로 데이터를 부분집합으로 분할합니다. 이 부분집합들은 윈도우 함수의 계산 범위를 정의하며, 각 부분집합 내에서 독립적으로 함수가 계산됩니다.
    - **사용 예**: `PARTITION BY column1, column2` 의 경우, `column1` 과 `column2` 의 값 조합이 같은 데이터 행들이 같은 그룹(부분집합)을 형성합니다. 이 그룹들은 윈도우 함수의 적용 대상이 됩니다.

```
FUNCTION() OVER (PARTITION BY column1, column2, ...)
```

2. **ORDER BY**

- **역할**: `ORDER BY` 는 각 부분집합 내에서 데이터 행들의 정렬 순서를 지정합니다. 이는 순위나 누적 합계 같은 윈도우 함수의 결과에 직접적인 영향을 미칩니다.
    - **사용 예**: 윈도우 함수와 함께 사용될 때, `ORDER BY` 는 데이터를 특정 기준(예: 날짜, 금액 등)에 따라 정렬하여 그 순서대로 함수가 계산되게 합니다.

```
FUNCTION() OVER (PARTITION BY column1, column2 ... ORDER BY column3
[ASC|DESC])
```

3. **ROWS/RANGE**

- **역할**: `ROWS/RANGE` 는 `ORDER BY` 와 함께 사용되어, 윈도우 함수가 특정 행의 범위 내에서만 데이터를 계산하도록 지정합니다. `ROWS` 는 물리적 행의 위치를 기준으로 범위를 설정하고, `RANGE` 는 정렬 키의 값에 따라 범위를 설정합니다.
    - **사용 예**: `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` 는 현재 행을 중심으로 이전 행과 다음 행을 포함하여 데이터를 계산합니다. `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 는 시작부터 현재 행까지의 데이터를 사용합니다.

ROWS 예시:

- `UNBOUNDED PRECEDING` : 파티션의 첫 행부터 시작
- `UNBOUNDED FOLLOWING` : 파티션의 마지막 행까지
- `CURRENT ROW` : 현재 행 포함
- `n PRECEDING/FOLLOWING` : 현재 행에서 n행 앞이나 뒤

```
FUNCTION() OVER (PARTITION BY column1, column2 ... ORDER BY column3
                 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

- **RANGE 예시**:

```
FUNCTION() OVER (PARTITION BY column1, column2 ... ORDER BY column3
                 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

sakila 데이터베이스를 이용한 예제

예제 1: 고객별, 대여 날짜별 누적 대여 횟수 계산

다음 쿼리는 `rental` 테이블에서 각 고객별, 대여 날짜별 누적 대여 횟수를 계산합니다.

```
SELECT
    customer_id,
    rental_date,
    COUNT(*) OVER (PARTITION BY customer_id ORDER BY rental_date
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_rentals
FROM
    rental
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| customer_id | rental_date          | cumulative_rentals |
|-------------|----------------------|--------------------|
| 1           | 2005-05-24 22:53:30  | 1                  |
| 1           | 2005-05-24 22:55:23  | 2                  |
| 1           | 2005-05-24 22:57:20  | 3                  |
| 1           | 2005-05-24 22:59:15  | 4                  |
| ...         | ...                  | ...                |
```

- 위 구문은 각 행에 대해 정의된 윈도우(여기서는 각 고객별로, 그리고 렌탈 날짜 순으로) 내에서 `COUNT(*)` 를 계산합니다.
- `PARTITION BY customer_id` 는 결과를 각 고객별로 분리하여 집계하라는 것을 의미합니다.
- `ORDER BY rental_date` 는 각 고객의 렌탈 데이터를 날짜 순서대로 정렬하라는 것을 지시합니다.
- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 는 각 행에 대해, 그 행을 포함하여 그 이전의 모든 행을 윈도우의 범위로 설정합니다.
  - 현재 행까지의 모든 데이터를 포함하여 COUNT(*) 를 계산합니다.
- 결과적으로 `cumulative_rentals` 값은 각 고객이 첫 번째 렌탈부터 현재 렌탈까지 누적하여 렌탈한 횟수를 나타냅니다.
  - 첫 번째 행에서는 렌탈 1회를, 두 번째 행에서는 렌탈 2회를, 이런 식으로 계속 증가합니다.

FUNCTION()

- 다음 구문의 `COUNT(*)` 에는 다양한 함수가 적용될 수 있음
  - **COUNT()** - 특정 범위의 데이터 행의 수를 계산합니다.
  2. **SUM(expression)** - 지정된 표현식의 합계를 계산합니다.
  3. **AVG(expression)** - 지정된 표현식의 평균 값을 계산합니다.
  4. **MIN(expression)** - 지정된 표현식의 최소 값을 찾습니다.
  5. **MAX(expression)** - 지정된 표현식의 최대 값을 찾습니다.
  6. **ROW_NUMBER()** - 현재 행의 순번을 반환합니다.
  7. **RANK()** - 현재 윈도우 내에서 현재 행의 순위를 반환합니다 (동점을 고려하여 순위가 건너뛰어짐).
  8. **DENSE_RANK()** - 현재 윈도우 내에서 현재 행의 순위를 반환합니다 (동점을 고려하여 순위가 건너뛰어지지 않음).
  9. **LEAD(expression, offset, default)** - 지정된 행보다 앞에 위치한 행의 데이터를 반환합니다.
  10. **LAG(expression, offset, default)** - 지정된 행보다 뒤에 위치한 행의 데이터를 반환합니다.
  11. **FIRST_VALUE(expression)** - 윈도우 내 첫 번째 값 반환.
  12. **LAST_VALUE(expression)** - 윈도우 내 마지막 값 반환.

```
    COUNT(*) OVER (PARTITION BY customer_id ORDER BY rental_date
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_rentals
```

SUM() 예제

```
SELECT
    R.customer_id,
    R.rental_date,
    P.amount,
    SUM(P.amount) OVER (PARTITION BY R.customer_id ORDER BY R.rental_date
                        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, R.rental_date
```

AVG() 예제:

```
SELECT
    R.customer_id,
    R.rental_date,
    P.amount,
    AVG(P.amount) OVER (PARTITION BY R.customer_id ORDER BY R.rental_date
                        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, R.rental_date
```

이 예제는 각 고객이 첫 렌탈부터 현재 렌탈까지의 평균 렌탈 비용을 보여줍니다.

RANGE 와 ROWS

```
SELECT
    R.customer_id,
    R.rental_date,
    P.amount,
    SUM(P.amount) OVER (PARTITION BY R.customer_id ORDER BY R.rental_date
                        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, R.rental_date


SELECT
    R.customer_id,
    R.rental_date,
    P.amount,
    SUM(P.amount) OVER (PARTITION BY R.customer_id ORDER BY R.rental_date
                        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, R.rental_date
```

윈도우 함수에서 `ROWS` 와 `RANGE` 는 윈도우의 범위를 정의하는 데 사용되는 두 가지 접근 방식입니다. 이들의 차이점을 이해하려면 먼저 각 방식이 데이터를 어떻게 그룹화하고 집계하는지 알아야 합니다. 여기서는 두 쿼리를 비교하면서 `ROWS` 와 `RANGE` 의 차이점을 설명하겠습니다.

ROWS

- `ROWS` 키워드는 물리적인 행의 위치를 기준으로 윈도우를 정의합니다.
- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 는 현재 행을 포함하여 그 이전의 모든 행들을 집계 범위로 삼습니다.
- 이 방식은 각 행을 고유하게 취급하며, 정렬된 순서에 따라 정확하게 해당 위치의 행들만을 집계에 포함합니다.

RANGE

- `RANGE` 키워드는 현재 행의 정렬 기준 값(`ORDER BY` 에 지정된 값)과 동일한 값을 가진 모든 행을 윈도우에 포함합니다.
- `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 는 현재 행의 정렬 키 값과 동일한 모든 행을 포함하여 그 이전의 모든 행들까지를 집계 범위로 삼습니다.
- 만약 정렬 키 값이 중복되는 경우, `RANGE` 는 그 값에 해당하는 모든 행을 같은 그룹으로 간주하여 집계합니다. 이는 `ROWS` 방식에서는 볼 수 없는 특징입니다.

데이터로 이해하기

고객의 결제 데이터를 다루는 상황을 가정해보겠습니다. 고객이 다음과 같은 결제 기록을 가지고 있다고 가정합니다:

Customer ID	Rental Date	Amount
1	2021-07-01	$5
1	2021-07-01	$3
1	2021-07-02	$7
1	2021-07-03	$6

ROWS를 사용하는 쿼리 결과:

- 첫 번째 및 두 번째 행은 동일한 날짜(2021-07-01)를 가지지만, `ROWS` 는 각 행을 독립적으로 계산합니다.
- 결과적으로, 두 번째 행의 누적 금액은 첫 번째 행의 금액을 포함하여 $8이 됩니다.

RANGE를 사용하는 쿼리 결과:

- `RANGE` 는 같은 날짜(2021-07-01)에 발생한 모든 행을 동일한 그룹으로 봅니다.
- 따라서, 첫 번째와 두 번째 행 모두 누적 금액이 $8이 됩니다.

이 차이는 특히 결제 데이터에서 같은 날 여러 거래가 발생할 때 중요합니다. `RANGE` 를 사용하면 같은 날의 모든 결제를 한 거래처럼 취급하여 집계하므로, 일부 데이터 분석에서는 이 방식이 더 적합할 수 있습니다. 반면, `ROWS` 는 각 행을 독립적인 사건으로 간주하여 더 세분화된 데이터 분석을 가능하게 합니다.

예제로 이해하기

- 다음 쿼리는 RANGE 를 사용하였지만, rental_date 는 동일 날짜라 하더라도, 시간이 다르기 때문에 동일 값을 가진 행으로 인식하지 않음

```
SELECT
    R.customer_id,
    R.rental_date,
    P.amount,
    SUM(P.amount) OVER (PARTITION BY R.customer_id ORDER BY R.rental_date
                        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, R.rental_date
```

- 다음과 같이 rental_date 에서 날짜만 빼와서 이를 기반으로 정렬하도록 변경하면, RANGE 를 사용했으므로, 동일 값을 가진 행으로 인식하여 계산됨

```
SELECT
    R.customer_id,
    R.rental_date,
    DATE(R.rental_date),
    P.amount,
    SUM(P.amount) OVER (
        PARTITION BY R.customer_id
        ORDER BY DATE(R.rental_date) -- DATE 함수를 사용하여 날짜 부분만 추출합니
다.
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_payment_amount
FROM
    payment P
JOIN
    rental R ON P.rental_id = R.rental_id
ORDER BY
    R.customer_id, DATE(R.rental_date) -- 여기도 날짜만 사용합니다.
```

예제 2: 영화별 누적 대여 수익 계산

다음 쿼리는 `payment` , `rental` , `inventory` 테이블을 조인하여 각 영화별 누적 대여 수익을 계산합니다.

```
SELECT DISTINCT
    i.film_id,
    p.amount,
    SUM(p.amount) OVER (PARTITION BY i.film_id ORDER BY p.payment_date
                        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
AS cumulative_revenue
FROM
    payment p
JOIN
    rental r ON p.rental_id = r.rental_id
JOIN
    inventory i ON r.inventory_id = i.inventory_id
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| film_id | cumulative_revenue |
|---------|--------------------|
| 1       | 20.97              |
| 1       | 26.97              |
| 1       | 32.97              |
| 1       | 41.97              |
| ...     | ...                |
```

여기서 `PARTITION BY i.film_id` 는 윈도우를 영화별로 나누고, `ORDER BY p.payment_date` 는 각 파티션 내에서 결제 날짜순으로 행을 정렬합니다. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 는 각 행마다 파티션의 첫 번째 행부터 현재 행까지의 윈도우를 정의합니다.

예제 3: 영화 대여 내역에서 고객별, 날짜별 누적 대여 횟수 계산

영화 대여 업체에서는 고객의 대여 패턴을 분석하기 위해, 각 고객이 시간 경과에 따라 몇 편의 영화를 대여했는지 누적 대여 횟수를 계산하고자 합니다.

```
SELECT
    r.customer_id,
    DATE(r.rental_date) AS rental_date,
    COUNT(*) OVER (
        PARTITION BY r.customer_id
        ORDER BY DATE(r.rental_date)
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_rentals
FROM
    rental r
ORDER BY
    r.customer_id, DATE(r.rental_date);
```

이 쿼리에서는 `PARTITION BY`를 사용하여 데이터를 고객별로 분할하고, `ORDER BY`를 사용하여 각 고객 내에서 대여 날짜순으로 정렬합니다. 그리고 `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`를 사용하여, 각 행마다 해당 고객의 첫 번째 대여부터 **현재 대여 날짜**까지의 누적 대여 횟수를 계산합니다.

예제 4: 장르별 영화 대여 수익 분석

영화 대여 업체에서는 각 영화 장르의 수익성을 분석하기 위해, 장르별 대여 수익의 누적 합계와 전체 대여 수익 대비 비율을 계산하고자 합니다.

```
WITH genre_revenue AS (
    SELECT
        c.name AS genre,
        SUM(p.amount) AS revenue
    FROM
        payment p
        JOIN rental r ON p.rental_id = r.rental_id
        JOIN inventory i ON r.inventory_id = i.inventory_id
        JOIN film f ON i.film_id = f.film_id
        JOIN film_category fc ON f.film_id = fc.film_id
        JOIN category c ON fc.category_id = c.category_id
    GROUP BY
        c.name
)
SELECT
    genre,
    revenue,
    SUM(revenue) OVER (
        ORDER BY revenue DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

```
    ) AS cumulative_revenue,
    revenue / SUM(revenue) OVER () AS revenue_ratio
FROM
    genre_revenue
ORDER BY
    revenue DESC;
```

이 쿼리에서는 먼저 `genre_revenue` CTE(Common Table Expression)를 사용하여 장르별 대여 수익을 계산합니다. 그리고 주 쿼리에서 `ORDER BY` 를 사용하여 수익 내림차순으로 정렬하고, `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 를 사용하여 각 행까지의 누적 수익을 계산합니다. 마지막으로 `SUM(revenue) OVER ()` 를 사용하여 전체 대여 수익을 계산하고, 이를 기준으로 각 장르의 수익 비율을 계산합니다.

`SUM(revenue)` 와 `SUM(revenue) OVER ()` 차이

> 단순하게, 결과가 여러 행이면, 각 행마다 계산이 되어야 하므로 이 때는 SUM(revenue) OVER () 를 사용해야 하고, 결과가 한 행일 때만 SUM(revenue) 가 (GROUP BY 없이) 단독 사용이 가능함

1. **`SUM(revenue)`** :
    - 이것은 일반적인 집계 함수로, 전체 쿼리 결과 또는 그룹화된 데이터 세트의 합계를 계산합니다.
    - `GROUP BY` 절과 함께 사용될 경우, 각 그룹의 `revenue` 합계를 계산합니다.
    - `GROUP BY` 절이 없는 경우 전체 쿼리 결과의 `revenue` 합계를 반환합니다.
2. **`SUM(revenue) OVER ()`** :
    - 이 표현은 윈도우 함수로, 'OVER' 절을 사용하여 정의됩니다. 여기서 `OVER ()` 는 특정 파티션 또는 정렬 없이 전체 결과 세트에 대해 함수를 적용하라는 의미입니다.
    - 윈도우 함수로서 `SUM(revenue) OVER ()` 는 쿼리가 반환하는 각 행에 대해 전체 결과 집합의 `revenue` 합계를 계산합니다. 이는 모든 행에서 같은 값을 반환하며, 각 행은 전체 `revenue` 의 합계를 알 수 있습니다.

위 쿼리에서 `SUM(revenue) OVER ()` 는 `genre_revenue` 에서 계산된 각 장르별 매출 (`revenue`)의 전체 합을 모든 행에 동일하게 반환하여, 각 장르가 전체 매출에서 차지하는 비율 (`revenue_ratio`)을 계산하는 데 사용됩니다. 이것은 각 행마다 전체 매출 대비 해당 장르 매출의 비율을 보여줍니다.

- 다음 쿼리같이 단일 행을 출력할 때는 SUM(revenue) 가능

```
WITH genre_revenue AS (
    SELECT
        c.name AS genre,
        SUM(p.amount) AS revenue
    FROM
        payment p
        JOIN rental r ON p.rental_id = r.rental_id
```

```
        JOIN inventory i ON r.inventory_id = i.inventory_id
        JOIN film f ON i.film_id = f.film_id
        JOIN film_category fc ON f.film_id = fc.film_id
        JOIN category c ON fc.category_id = c.category_id
    GROUP BY
        c.name
)
SELECT SUM(revenue) FROM genre_revenue
```

- 다음 쿼리같이 여러 행을 출력할 때는 정상 동작할 수 없음

```
WITH genre_revenue AS (
    SELECT
        c.name AS genre,
        SUM(p.amount) AS revenue
    FROM
        payment p
        JOIN rental r ON p.rental_id = r.rental_id
        JOIN inventory i ON r.inventory_id = i.inventory_id
        JOIN film f ON i.film_id = f.film_id
        JOIN film_category fc ON f.film_id = fc.film_id
        JOIN category c ON fc.category_id = c.category_id
    GROUP BY
        c.name
)
SELECT revenue / SUM(revenue) FROM genre_revenue
```

- 이 때는 다음과 같이 윈도우 함수를 사용해야 함 (각 행에 적용 가능함)

```
WITH genre_revenue AS (
    SELECT
        c.name AS genre,
        SUM(p.amount) AS revenue
    FROM
        payment p
        JOIN rental r ON p.rental_id = r.rental_id
        JOIN inventory i ON r.inventory_id = i.inventory_id
        JOIN film f ON i.film_id = f.film_id
        JOIN film_category fc ON f.film_id = fc.film_id
        JOIN category c ON fc.category_id = c.category_id
    GROUP BY
```

```
        c.name
)
SELECT revenue / SUM(revenue) OVER () FROM genre_revenue
```

> 윈도우 함수는 각 행의 컨텍스트에서 전체 또는 부분 집합의 데이터를 참조하여 계산을 수행할 필요가 있을 때 사용됨!

LEAD(), LAG(), FIRST_VALUE(), LAST_VALUE() 문법

SQL에서 `LEAD()`, `LAG()`, `FIRST_VALUE()`, `LAST_VALUE()` 함수는 행 간의 관계를 기반으로 값을 가져오는 데 사용되는 윈도우 함수입니다.

1. `LEAD(column, n, default)` : 현재 행을 기준으로 n행 뒤의 값을 가져옵니다. n을 지정하지 않으면 기본값은 1입니다. 뒤에 행이 없으면 default 값을 반환합니다.

```
LEAD(column, n, default) OVER (ORDER BY column_name [ASC|DESC])
```

2. `LAG(column, n, default)` : 현재 행을 기준으로 n행 앞의 값을 가져옵니다. n을 지정하지 않으면 기본값은 1입니다. 앞에 행이 없으면 default 값을 반환합니다.

```
LAG(column, n, default) OVER (ORDER BY column_name [ASC|DESC])
```

3. `FIRST_VALUE(column)` : 파티션된 윈도우에서 첫 번째 값을 가져옵니다.

```
FIRST_VALUE(column) OVER (PARTITION BY column_name ORDER BY column_name
[ASC|DESC])
```

4. `LAST_VALUE(column)` : 파티션된 윈도우에서 마지막 값을 가져옵니다.

```
LAST_VALUE(column) OVER (PARTITION BY column_name ORDER BY column_name
[ASC|DESC]
                         RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
FOLLOWING)
```

이 함수들은 모두 `OVER` 절과 함께 사용되며, `ORDER BY` 절을 통해 윈도우 내의 행 순서를 정의합니다. `FIRST_VALUE()` 와 `LAST_VALUE()` 는 추가로 `PARTITION BY` 절을 사용하여 윈도우를 파티션으로 나눌 수 있습니다.

sakila 데이터베이스를 이용한 예제

`sakila` 데이터베이스를 사용하여 이 함수들의 사용법을 살펴보겠습니다.

예제 1: 영화 대여 내역에서 이전 대여와 다음 대여 찾기

다음 쿼리는 `rental` 테이블에서 각 대여 내역의 이전 대여와 다음 대여의 `rental_id` 를 찾습니다.

```
SELECT
    rental_id,
    rental_date,
    LAG(rental_id, 1, 0) OVER (ORDER BY rental_date) AS prev_rental,
    LEAD(rental_id, 1, 0) OVER (ORDER BY rental_date) AS next_rental
FROM
    rental
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| rental_id | rental_date           | prev_rental | next_rental |
|-----------|-----------------------|-------------|-------------|
| 1         | 2005-05-24 22:53:30   | 0           | 2           |
| 2         | 2005-05-24 22:54:33   | 1           | 3           |
| 3         | 2005-05-24 23:03:39   | 2           | 4           |
| 4         | 2005-05-24 23:04:41   | 3           | 5           |
| ...       | ...                   | ...         | ...         |
```

여기서 `LAG()` 는 이전 대여의 `rental_id` 를 가져오고, `LEAD()` 는 다음 대여의 `rental_id` 를 가져옵니다. 첫 번째 대여의 이전 대여와 마지막 대여의 다음 대여는 존재하지 않으므로, default 값인 0이 사용되었습니다.

예제 2: 영화별 첫 번째와 마지막 대여 날짜 찾기

다음 쿼리는 `rental` 과 `inventory` 테이블을 조인하여 각 영화(`film_id`)별로 첫 번째 대여 날짜와 마지막 대여 날짜를 찾습니다.

```
SELECT DISTINCT
    i.film_id,
    FIRST_VALUE(r.rental_date) OVER (PARTITION BY i.film_id ORDER BY
r.rental_date) AS first_rental,
    LAST_VALUE(r.rental_date) OVER (PARTITION BY i.film_id ORDER BY
r.rental_date
                                    ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING) AS last_rental
FROM
    rental r
JOIN
    inventory i ON r.inventory_id = i.inventory_id
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| film_id | first_rental          | last_rental           |
|---------|-----------------------|-----------------------|
| 1       | 2005-05-24 22:53:30   | 2006-02-14 15:16:03   |
| 2       | 2005-05-24 22:54:33   | 2006-01-31 22:12:39   |
| 3       | 2005-05-24 23:03:39   | 2005-08-23 17:32:07   |
| ...     | ...                   | ...                   |
```

여기서 `FIRST_VALUE()` 는 각 영화(`film_id`)별로 첫 번째 대여 날짜를 가져오고, `LAST_VALUE()` 는 마지막 대여 날짜를 가져옵니다. `PARTITION BY` 절은 윈도우를 영화별로 나누고, `ORDER BY` 절은 윈도우 내에서 대여 날짜순으로 행을 정렬합니다.

- 다음 쿼리가 정상동작하지 않는 이유
  - 윈도우 함수에 `RANGE` 또는 `ROWS` 구문이 명시되지 않으면, 디폴트로 다음 구문이 설정됨
    - `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
    - 이로 인해 각 파티션에서 현재 행까지에 대해서 LAST_VALUE() 값을 계산하므로, 전체 파티션의 값을 기반으로 계산되지 않음

```
SELECT DISTINCT
    i.film_id,
    FIRST_VALUE(r.rental_date) OVER (PARTITION BY i.film_id ORDER BY
r.rental_date) AS first_rental,
    LAST_VALUE(r.rental_date) OVER (PARTITION BY i.film_id ORDER BY
r.rental_date) AS last_rental
FROM
    rental r
JOIN
    inventory i ON r.inventory_id = i.inventory_id
```

예제 3: 영화 대여 내역에서 고객별, 대여 순서별 대여 간격 계산

영화 대여 업체에서는 고객의 대여 주기를 분석하기 위해, 각 고객의 연속된 대여 사이의 시간 간격을 계산하고자 합니다.

```
SELECT
    r.customer_id,
    r.rental_id,
    r.rental_date,
    DATEDIFF(
        r.rental_date,
        LAG(r.rental_date) OVER (
            PARTITION BY r.customer_id
            ORDER BY r.rental_date
        )
    ) AS days_since_last_rental
FROM
    rental r
ORDER BY
    r.customer_id, r.rental_date;
```

이 쿼리에서는 `PARTITION BY` 를 사용하여 데이터를 고객별로 분할하고, `ORDER BY` 를 사용하여 각 고객 내에서 대여 날짜순으로 정렬합니다. 그리고 `LAG` 함수를 사용하여 각 대여의 바로 이전 대여 날짜를 가져오고, `DATEDIFF` 함수를 사용하여 현재 대여와 이전 대여 사이의 일수 차이를 계산합니다.

PERCENT_RANK(), CUME_DIST(), NTILE() 문법

SQL에서 `PERCENT_RANK()`, `CUME_DIST()`, `NTILE()` 함수는 행의 상대적인 위치를 기반으로 값을 계산하는 데 사용되는 윈도우 함수입니다.

1. `PERCENT_RANK()` : 행의 백분위 순위를 계산합니다. 결과는 0부터 1 사이의 값입니다.

```
PERCENT_RANK() OVER (ORDER BY column_name [ASC|DESC])
```

2. `CUME_DIST()` : 행의 누적 분포를 계산합니다. 결과는 0부터 1 사이의 값입니다.

```
CUME_DIST() OVER (ORDER BY column_name [ASC|DESC])
```

3. `NTILE(n)` : 행을 n개의 그룹으로 분할합니다. 각 그룹에는 거의 같은 수의 행이 포함됩니다.

```
NTILE(n) OVER (ORDER BY column_name [ASC|DESC])
```

이 함수들은 모두 `OVER` 절과 함께 사용되며, `ORDER BY` 절을 통해 윈도우 내의 행 순서를 정의합니다.

sakila 데이터베이스를 이용한 예제

`sakila` 데이터베이스를 사용하여 이 함수들의 사용법을 살펴보겠습니다.

예제 1: 영화 길이에 따른 백분위 순위와 누적 분포 계산

다음 쿼리는 `film` 테이블에서 각 영화의 길이(`length`)에 따른 백분위 순위와 누적 분포를 계산합니다.

```
SELECT
    title,
    length,
    PERCENT_RANK() OVER (ORDER BY length) AS percent,
    CUME_DIST() OVER (ORDER BY length) AS cume
FROM
    film
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| title           | length | percent_rank | cume_dist |
|-----------------|--------|--------------|-----------|
| Short Film 1    | 46     | 0            | 0.0054    |
| Short Film 2    | 46     | 0            | 0.0054    |
| Short Film 3    | 50     | 0.0108       | 0.0162    |
| Short Film 4    | 50     | 0.0108       | 0.0162    |
| ...             | ...    | ...          | ...       |
```

여기서 `PERCENT_RANK()` 는 각 영화의 길이에 따른 백분위 순위를 계산합니다. 가장 짧은 영화는 0, 가장 긴 영화는 1에 가까운 값을 가집니다. `CUME_DIST()` 는 각 영화의 길이에 따른 누적 분포를 계산합니다. 이는 해당 영화보다 짧거나 같은 영화의 비율을 나타냅니다.

데이터베이스에서 `PERCENT_RANK()` 와 `CUME_DIST()` 는 윈도우 함수로서, 각각 행의 상대적 위치와 누적 분포를 계산하는 데 사용됩니다. 이 두 함수는 특히 데이터를 정렬 순서에 따라 분석할 때 유용합니다. 다음은 `film` 테이블의 `title` 과 `length` 를 사용하여 이 함수들이 어떻게 작동하는지에 대한 상세한 설명입니다. 예를 들어, 몇 개의 영화 데이터가 다음과 같이 주어진다고 가정해 보겠습니다:

title	length
Movie A	90
Movie B	120
Movie C	90
Movie D	150

PERCENT_RANK() 계산

`PERCENT_RANK()` 함수는 주어진 행의 퍼센트 순위를 계산합니다. 이 순위는 `(rank - 1) / (전체 행 수 - 1)` 공식을 사용하여 계산되며, 여기서 `rank` 는 `ORDER BY` 절에 따라 할당된 행의 순위입니다.

1. **Movie A**와 **Movie C**는 길이가 90분으로 동일합니다. 이들은 순위 공유를 하므로, 최소 순위인 1을 받습니다.
2. **Movie B**의 길이는 120분으로 다음으로 긴 길이입니다. 세 번째 순위를 받습니다.
3. **Movie D**는 가장 긴 길이인 150분을 가지고 있으며, 이는 네 번째 순위입니다.

그래서 각 영화의 `PERCENT_RANK()` 는 다음과 같이 계산됩니다:

- **Movie A**와 **Movie C**: `(1 - 1) / (4 - 1) = 0`
- **Movie B**: `(3 - 1) / (4 - 1) = 0.6667`
- **Movie D**: `(4 - 1) / (4 - 1) = 1.0`

CUME_DIST() 계산

`CUME_DIST()` 함수는 주어진 행의 누적 분포를 계산합니다. 이는 `주어진 값보다 작거나 같은 행의 수 / 전체 행 수` 공식을 사용하여 계산됩니다.

1. **Movie A**와 **Movie C**는 길이가 90분입니다. 이들은 전체 데이터의 2/4 또는 50%를 차지합니다.
2. **Movie B**는 120분으로, 길이가 90분 이하인 영화를 포함하여 전체의 3/4 또는 75%를 차지합니다.
3. **Movie D**는 가장 길어서, 모든 영화를 포함하여 100% 또는 전체를 차지합니다.

그래서 각 영화의 `CUME_DIST()` 는 다음과 같이 계산됩니다:

- **Movie A**와 **Movie C**: `2 / 4 = 0.5`
- **Movie B**: `3 / 4 = 0.75`
- **Movie D**: `4 / 4 = 1.0`

SQL 쿼리 결과

결과적으로, SQL 쿼리의 결과는 다음과 같을 것입니다:

title	length	percent	cume
Movie A	90	0.0000	0.5000
Movie C	90	0.0000	0.5000
Movie B	120	0.6667	0.7500
Movie D	150	1.0000	1.0000

이 예제를 통해 `PERCENT_RANK()` 와 `CUME_DIST()` 함수가 어떻게 각 행의 위치와 누적 분포를 계산하는지 이해할 수 있습니다.

예제 2: 고객을 4개의 그룹으로 분할

다음 쿼리는 `customer` 테이블에서 고객을 4개의 그룹으로 분할합니다.

```sql
SELECT
    customer_id,
    CONCAT(first_name, ' ', last_name) AS customer_name,
    NTILE(4) OVER (ORDER BY customer_id) AS customer_group
FROM
    customer
```

이 쿼리의 결과는 다음과 같을 수 있습니다.

```
| customer_id | customer_name | customer_group |
|-------------|---------------|----------------|
| 1           | MARY SMITH    | 1              |
| 2           | PATRICIA JOHNSON | 1           |
| 3           | LINDA WILLIAMS| 1              |
| 4           | BARBARA JONES | 1              |
| 5           | ELIZABETH BROWN | 2            |
| 6           | JENNIFER DAVIS| 2              |
| 7           | MARIA MILLER  | 2              |
| 8           | SUSAN WILSON  | 2              |
| 9           | MARGARET MOORE| 3              |
| 10          | DOROTHY TAYLOR| 3              |
```

여기서 `NTILE(4)` 는 고객을 4개의 그룹으로 분할합니다. 각 그룹에는 거의 같은 수의 고객이 포함됩니다. 고객 ID순으로 정렬되었기 때문에, ID가 작은 고객은 낮은 그룹에, ID가 큰 고객은 높은 그룹에 속하게 됩니다.

RANK() / DENSE_RANK() 연습 문제 1

고객(customer) 테이블에서 각 고객의 총 지출 금액(total_spent)을 계산하고, 총 지출 금액에 따라 고객의 순위를 매기세요. 결과에는 고객 ID(customer_id), 고객 이름(first_name, last_name), 총 지출 금액(total_spent), 순위(rank)가 포함되어야 합니다.

```sql
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    SUM(p.amount) AS total_spent,
    RANK() OVER (ORDER BY SUM(p.amount) DESC) AS ranking
FROM
    customer c
    JOIN payment p ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id
ORDER BY
    ranking;
```

RANK() / DENSE_RANK() 연습 문제 2

영화(film) 테이블에서 각 영화의 대여 횟수(rental_count)를 계산하고, 대여 횟수에 따라 영화의 순위를 매기세요. 같은 대여 횟수의 영화는 같은 순위를 가져야 하지만, 다음 순위는 건너뛰지 않습니다. 결과에는 영화 제목(title), 대여 횟수(rental_count), 순위(rank)가 포함되어야 합니다.

```sql
SELECT
  F.title,
    COUNT(*) rental_count,
    DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) 'rank'
FROM film F
JOIN inventory I USING(film_id)
JOIN rental R USING(inventory_id)
GROUP BY F.film_id
```

LEAD() / LAG() 연습 문제 1

대여(rental) 테이블에서 각 대여의 이전 대여와 다음 대여의 대여 ID(rental_id)를 출력하세요. 결과에는 대여 ID(rental_id), 대여 일자(rental_date), 이전 대여 ID(previous_rental_id), 다음 대여 ID(next_rental_id)가 포함되어야 합니다.

```sql
SELECT
    rental_id,
    rental_date,
    LAG(rental_id) OVER (ORDER BY rental_date) AS previous_rental_id,
    LEAD(rental_id) OVER (ORDER BY rental_date) AS next_rental_id
FROM
    rental;
```

LEAD() / LAG() 연습 문제 2

지불(payment) 테이블에서 각 지불에 대해 같은 고객의 이전 지불 금액과 다음 지불 금액을 출력하세요. 결과에는 지불 ID(payment_id), 고객 ID(customer_id), 지불 금액(amount), 이전 지불 금액(previous_amount), 다음 지불 금액(next_amount)이 포함되어야 합니다.

```sql
SELECT
    payment_id,
    customer_id,
    amount,
    LAG(amount) OVER (PARTITION BY customer_id ORDER BY payment_date) AS
previous_amount,
    LEAD(amount) OVER (PARTITION BY customer_id ORDER BY payment_date) AS
next_amount
FROM
    payment;
```

FIRST_VALUE() / LAST_VALUE() 연습 문제 1

대여(rental) 테이블에서 각 고객(customer_id)별로 첫 번째 대여 일자와 마지막 대여 일자를 출력하세요. 결과에는 고객 ID(customer_id), 첫 번째 대여 일자(first_rental_date), 마지막 대여 일자(last_rental_date)가 포함되어야 합니다.

```sql
SELECT DISTINCT
    customer_id,
    FIRST_VALUE(rental_date) OVER (PARTITION BY customer_id ORDER BY
rental_date) AS first_rental_date,
    LAST_VALUE(rental_date) OVER (PARTITION BY customer_id ORDER BY
rental_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
last_rental_date
FROM
    rental;
```

FIRST_VALUE() / LAST_VALUE() 연습 문제 2
지불(payment) 테이블에서 각 직원(staff_id)이 처리한 첫 번째 지불과 마지막 지불의 금액을 출력하세요. 결과에는 직원 ID(staff_id), 첫 번째 지불 금액(first_payment_amount), 마지막 지불 금액(last_payment_amount)이 포함되어야 합니다.

```sql
SELECT DISTINCT
    staff_id,
    FIRST_VALUE(amount) OVER (PARTITION BY staff_id ORDER BY payment_date)
AS first_payment_amount,
    LAST_VALUE(amount) OVER (PARTITION BY staff_id ORDER BY payment_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
last_payment_amount
FROM
    payment;
```

PERCENT_RANK() / CUME_DIST() 연습 문제 1
영화(film) 테이블에서 각 영화의 대여 기간(rental_duration)에 대한 백분위 순위와 누적 분포를 계산하세요. 결과에는 영화 제목(title), 대여 기간(rental_duration), 백분위 순위(percentile_rank), 누적 분포(cumulative_distribution)가 포함되어야 합니다.

```sql
SELECT
    title,
    rental_duration,
    PERCENT_RANK() OVER (ORDER BY rental_duration) AS percentile_rank,
    CUME_DIST() OVER (ORDER BY rental_duration) AS cumulative_distribution
FROM
    film;
```

PERCENT_RANK() / CUME_DIST() 연습 문제 2

고객(customer) 테이블에서 각 고객의 총 지출 금액(total_amount)에 대한 백분위 순위와 누적 분포를 계산하세요. 결과에는 고객 ID(customer_id), 총 지출 금액(total_amount), 백분위 순위(percentile_rank), 누적 분포(cumulative_distribution)가 포함되어야 합니다.

```sql
SELECT
    c.customer_id,
    SUM(p.amount) AS total_amount,
    PERCENT_RANK() OVER (ORDER BY SUM(p.amount)) AS percentile_rank,
    CUME_DIST() OVER (ORDER BY SUM(p.amount)) AS cumulative_distribution
FROM
    customer c
    JOIN payment p ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id;
```

PARTITION BY, ORDER BY, ROWS/RANGE 연습 문제 1

대여(rental) 테이블에서 각 고객(customer_id)별로 대여 순서에 따른 누적 대여 횟수를 출력하세요. 결과에는 대여 ID(rental_id), 고객 ID(customer_id), 대여 일자(rental_date), 누적 대여 횟수(cumulative_rentals)가 포함되어야 합니다.

```sql
SELECT
    rental_id,
    customer_id,
    rental_date,
    COUNT(*) OVER (PARTITION BY customer_id ORDER BY rental_date
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_rentals
FROM
    rental;
```

PARTITION BY, ORDER BY, ROWS/RANGE 연습 문제 2

지불(payment) 테이블에서 각 고객(customer_id)별로 지불 일자(payment_date)에 따른 누적 지불 금액을 출력하세요. 결과에는 지불 ID(payment_id), 고객 ID(customer_id), 지불 일자(payment_date), 지불 금액(amount), 누적 지불 금액(cumulative_amount)이 포함되어야 합니다.

```sql
SELECT
    payment_id,
    customer_id,
    payment_date,
    amount,
    SUM(amount) OVER (PARTITION BY customer_id ORDER BY payment_date
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_amount
FROM
    payment;
```

PARTITION BY, ORDER BY, ROWS/RANGE 연습 문제 3

영화(film) 테이블에서 각 등급(rating)별로 (정렬은 film_id 를 기준으로 함) 영화 대여 기간(rental_duration)의 이동 평균을 출력하세요. 이동 평균은 현재 행과 이전 두 행을 포함하여 계산합니다. 결과에는 영화 ID(film_id), 등급(rating), 대여 기간(rental_duration), 이동 평균(moving_average)이 포함되어야 합니다.

```sql
SELECT
    film_id,
    rating,
    rental_duration,
    AVG(rental_duration) OVER (PARTITION BY rating ORDER BY film_id
                               ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
AS moving_average
FROM
    film;
```

PARTITION BY, ORDER BY, ROWS/RANGE 연습 문제 4

대여(rental) 테이블에서 각 직원(staff_id)의 대여 일자(rental_date)에 따른 대여 횟수와 각 직원(staff_id)별 누적 대여 횟수를 출력하세요. 결과에는 대여 ID(rental_id), 직원 ID(staff_id), 대여 일시(rental_date), 대여 횟수(rental_count), 누적 대여 횟수 (cumulative_rental_count)가 포함되어야 합니다.

```sql
SELECT
    rental_id,
    staff_id,
    rental_date,
    COUNT(*) OVER (PARTITION BY staff_id, DATE(rental_date)) AS
rental_count,
    COUNT(*) OVER (PARTITION BY staff_id ORDER BY rental_date
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_rental_count
FROM
    rental;
```

PARTITION BY, ORDER BY, ROWS/RANGE 연습 문제 5

고객(customer) 테이블과 지불(payment) 테이블을 사용하여 각 도시(city)별로 고객의 총 지불 금액 순위를 출력하세요. 순위는 도시 내에서 계산되어야 합니다. 결과에는 고객 ID(customer_id), 도시(city), 총 지불 금액(total_amount), 도시 내 순위(city_rank)가 포함되어야 합니다.

```sql
SELECT
    c.customer_id,
    ci.city,
    SUM(p.amount) AS total_amount,
    RANK() OVER (PARTITION BY ci.city ORDER BY SUM(p.amount) DESC) AS
city_rank
FROM
    customer c
    JOIN address a ON c.address_id = a.address_id
    JOIN city ci ON a.city_id = ci.city_id
    JOIN payment p ON c.customer_id = p.customer_id
GROUP BY
    c.customer_id;
```

NTILE() 연습 문제 1

고객(customer) 테이블에서 고객을 대여 횟수(rental_count)에 따라 4개의 그룹으로 나누세요. 결과에는 고객 ID(customer_id), 대여 횟수(rental_count), 그룹(ntile)이 포함되어야 합니다.

```sql
SELECT
  C.customer_id, COUNT(*) AS rental_count,
    NTILE(4) OVER (ORDER BY COUNT(*) DESC)
FROM customer C
JOIN rental R ON R.customer_id = C.customer_id
GROUP BY C.customer_id
```

NTILE() 연습 문제 2

영화(film) 테이블에서 영화를 대여 기간(rental_duration)에 따라 5개의 그룹으로 나누세요. 결과에는 영화 ID(film_id), 대여 기간(rental_duration), 그룹(ntile)이 포함되어야 합니다.

```sql
SELECT
  film_id, rental_duration,
    NTILE(5) OVER (ORDER BY rental_duration)
FROM film
```

ROW_NUMBER() 연습 문제 1

지불(payment) 테이블에서 각 고객(customer_id)별로 지불 내역에 행 번호를 부여하세요. (고객별 지불 내역의 행 번호는 payment_date 가 낮은순으로 부여하세요) 결과에는 지불 ID(payment_id), 고객 ID(customer_id), 지불 일자(payment_date), 지불 금액(amount), 행 번호(row_numbers)가 포함되어야 합니다.

```
SELECT
  payment_id, customer_id, payment_date, amount,
    ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY payment_date) AS
row_numbers
FROM payment
```

ROW_NUMBER() 연습 문제 2

영화(film) 테이블에서 각 등급(rating)별로 영화에 행 번호를 부여하세요. 영화는 대여 기간(rental_duration)에 따라 정렬되어야 합니다. 결과에는 영화 ID(film_id), 등급(rating), 대여 기간(rental_duration), 행 번호(row_number)가 포함되어야 합니다.

```
SELECT
  film_id, rating, rental_duration,
    ROW_NUMBER() OVER (PARTITION BY rating ORDER BY rental_duration) AS
row_numbers
FROM film
```

NTILE(), ROW_NUMBER() 연습 문제 3

고객(customer) 테이블과 지불(payment) 테이블을 사용하여 고객을 총 지불 금액(total_amount)에 따라 10개의 그룹으로 나누고, 각 그룹 내에서 고객별 총 지불 금액(total_amount)에 따라 행 번호를 부여하세요. 결과에는 고객 ID(customer_id), 총 지불 금액(total_amount), 그룹, 그룹 내 행 번호(row_number)가 포함되어야 합니다.

```
WITH CustomerPayments AS (
  SELECT  C.customer_id, SUM(P.amount) AS total_amount
    FROM customer C
    JOIN payment P ON P.customer_id = C.customer_id
    GROUP BY C.customer_id
),
```

```
    CustomerGroup AS (
      SELECT
        customer_id, total_amount,
            NTILE(10) OVER (ORDER BY total_amount) AS ten
        FROM CustomerPayments
    )
    SELECT customer_id, total_amount, ten,
      ROW_NUMBER() OVER (PARTITION BY ten ORDER BY total_amount) AS
    row_numbers
    FROM CustomerGroup
```