SQL 함수

SQL 함수는 값을 반환하는 내장 메소드입니다. 계산을 수행하거나 문자열을 조작하거나 날짜와 시간을 처리하거나 다른 데이터 조작을 수행하는 데 사용할 수 있습니다. 대부분의 SQL 함수는 데이터베이스 서버에 내장되어 있으므로 모든 SQL 쿼리에서 사용할 수 있습니다. 함수에는 여러 유형이 있습니다. 각 SQL 함수에 대한 설명과 예시를 `sakila` 데이터베이스를 기반으로 테스트해보며 알아보겠습니다.

1. 문자열 함수

`LENGTH(string)` : 문자열의 길이를 반환합니다.

- 예: `film` 테이블의 `title` 컬럼에 대한 각 영화 제목의 길이를 조회합니다.

```sql
SELECT title, LENGTH(title) AS title_length
FROM film
LIMIT 10;
```

`UPPER(string)` : 문자열을 대문자로 변환합니다.

- 예: `film` 테이블의 `title` 컬럼에 대한 영화 제목을 대문자로 변환하여 조회합니다.

```sql
SELECT UPPER(title) AS uppercased_title
FROM film
LIMIT 10;
```

`LOWER(string)` : 문자열을 소문자로 변환합니다.

- 예: `film` 테이블의 `title` 컬럼에 대한 영화 제목을 소문자로 변환하여 조회합니다.

```sql
SELECT LOWER(title) AS lowercased_title
FROM film
LIMIT 10;
```

`CONCAT(string1, string2, ...)` : 두 개 이상의 문자열을 하나로 연결합니다.

- 예: `actor` 테이블에서 `first_name` 과 `last_name` 을 연결하여 전체 이름을 조회합니다.

```sql
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM actor
LIMIT 10;
```

`SUBSTRING(string, start, length)` : 문자열에서 부분 문자열을 추출합니다.

- 예: `film` 테이블의 `description` 컬럼에서 첫 10글자를 추출합니다.

```sql
SELECT SUBSTRING(description, 1, 10) AS short_description
FROM film
LIMIT 10;
```

2. 날짜/시간 함수

`NOW()` : 현재 날짜와 시간을 반환합니다.

- 예: 현재 날짜와 시간을 조회합니다.

```sql
SELECT NOW() AS current_date_time;
```

`CURDATE()` : 현재 날짜를 반환합니다.

- 예: 현재 날짜를 조회합니다.

```
SELECT CURDATE() AS current_date;
```

`CURTIME()` : 현재 시간을 반환합니다.

- 예: 현재 시간을 조회합니다.

```
SELECT CURTIME() AS current_time;
```

`DATE_ADD(date, INTERVAL unit)` : 날짜에 간격을 추가합니다.

- 예: `rental` 테이블에서 각 대여 시작 날짜(`rental_date`)에 7일을 추가합니다.
- 년: YEAR, 달: MONTH, 일: DAY, 시간: HOUR, 분: MINUTE, 초: SECOND

```
SELECT rental_date, DATE_ADD(rental_date, INTERVAL 7 DAY) AS
return_date
FROM rental
LIMIT 10;
```

`DATE_SUB(date, INTERVAL unit)` : 날짜에서 간격을 뺍니다.

- 예: `rental` 테이블에서 각 대여 시작 날짜(`rental_date`)에서 7일을 뺍니다.
- 년: YEAR, 달: MONTH, 일: DAY, 시간: HOUR, 분: MINUTE, 초: SECOND

```
SELECT rental_date, DATE_SUB(rental_date, INTERVAL 7 DAY) AS
earlier_date
FROM rental
LIMIT 10;
```

EXTRACT(field FROM source)

- SQL의 `EXTRACT` 함수는 날짜 필드에서 특정 부분(예: 년, 월, 일 등)을 추출할 때 사용됩니다.
- `EXTRACT` 는 다음과 같은 구문을 가집니다

```
EXTRACT(field FROM source)
```

- `field` : 추출할 날짜의 부분 (YEAR, MONTH, DAY, HOUR, MINUTE 등)
- `source` : 날짜 값이 저장된 컬럼 또는 날짜 리터럴
- Sakila 데이터베이스 예제
  - `Sakila` 데이터베이스의 `payment` 테이블을 기반으로 몇 가지 예제를 들어보겠습니다. 이 테이블에는 `payment_date` 라는 날짜/시간 컬럼이 있습니다.

1. **모든 결제 레코드에서 년도만 추출하기**

```
SELECT EXTRACT(YEAR FROM payment_date) AS payment_year
FROM payment;
```

2. **각 월별 결제 횟수 세기**

```
SELECT EXTRACT(MONTH FROM payment_date) AS payment_month, COUNT(*)
FROM payment
GROUP BY payment_month;
```

- 세부 정보
  - `YEAR, MONTH, DAY` 는 달력 날짜를, `HOUR, MINUTE, SECOND` 는 시간을 추출합니다.

`YEAR()` , `MONTH()` , `DAY()` , HOUR(), MINUTE(), SECOND()

- `YEAR()` , `MONTH()` , `DAY()` 등의 함수는 `EXTRACT` 와 유사한 작업을 수행할 수 있습니다.
- 단, EXTRACT() 는 ANSI SQL(표준 SQL)에 들어 있는 함수이고, 위 함수들은 그렇지 않기 때문에, 일부 데이터베이스에서는 지원하지 않을 수 있음

1. **YEAR(), MONTH(), DAY()**

```
SELECT YEAR(payment_date) AS payment_year,
       MONTH(payment_date) AS payment_month,
       DAY(payment_date) AS payment_day
FROM payment;
```

2. **HOUR(), MINUTE(), SECOND()**

```
SELECT HOUR(payment_date) AS payment_hour,
       MINUTE(payment_date) AS payment_minute,
       SECOND(payment_date) AS payment_second
FROM payment;
```

3. 참고: **DAYOFWEEK()**

이 함수는 주어진 날짜가 일요일부터 시작하여 몇 번째 요일인지를 반환합니다 (일요일 = 1, 월요일 = 2, ... 토요일 = 7).

```
SELECT
    DAYOFWEEK(payment_date) AS payment_dayofweek, COUNT(*)
FROM payment
GROUP BY payment_dayofweek;
```

TIMESTAMPDIFF 기본 문법

- `TIMESTAMPDIFF` 함수는 두 날짜 또는 시간 값 사이의 차이를 계산합니다

```
TIMESTAMPDIFF(unit, start_datetime, end_datetime)
```

- `unit` : 반환할 시간 단위.
  - `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `YEAR` 등이 있습니다.
- `start_datetime` 과 `end_datetime` : 비교할 시작 및 종료 날짜/시간 값입니다.
- 예제
  `sakila` 데이터베이스의 `rental` 테이블을 사용해서 간단한 예제를 만들어볼게요. 이 테이블에는 `rental_date` 와 `return_date` 라는 두 개의 날짜/시간 컬럼이 있습니다.

1. **대여와 반납 사이의 일수 차이를 알고 싶다면:**

```
SELECT TIMESTAMPDIFF(DAY, rental_date, return_date) AS rental_days
FROM rental
LIMIT 5;
```

- 이 쿼리는 `rental` 테이블에서 5개의 레코드에 대해 대여일(`rental_date`)과 반납일(`return_date`) 사이의 차이를 일(`DAY`) 단위로 계산합니다.

2. **대여와 반납 사이의 시간 차이를 알고 싶다면:**

```
SELECT TIMESTAMPDIFF(HOUR, rental_date, return_date) AS rental_hours
FROM rental
LIMIT 5;
```

- 이 쿼리는 대여일과 반납일 사이의 차이를 시간(`HOUR`) 단위로 계산합니다.

DATE_FORMAT() 기본 문법

`DATE_FORMAT()` 은 MySQL에서 날짜 또는 시간 데이터를 특정 형식의 문자열로 변환하는 함수입니다. 이 함수는 다음과 같은 기본 구조를 가집니다.

```
DATE_FORMAT(date, format)
```

- `date` : 형식을 변경할 날짜 또는 시간 컬럼 또는 값
- `format` : 변환할 형식을 지정하는 문자열

형식 문자열에서 자주 사용되는 지시자 몇 가지:

- `%Y` : 4자리 연도
- `%y` : 2자리 연도
- `%M` : 월 이름
- `%m` : 2자리 월
- `%D` : 일 (영문 접미사 포함, 예: 2nd, 3rd, ...)
- `%d` : 2자리 일
- `%H` : 24시간 형식의 2자리 시간
- `%h` : 12시간 형식의 2자리 시간

- `%i` : 2자리 분
- `%s` : 2자리 초

sakila 데이터베이스를 이용한 간단한 예시

`sakila` 데이터베이스에는 `rental` 테이블에 `rental_date` 라는 날짜-시간 컬럼이 있습니다. 이를 이용해 `DATE_FORMAT()` 을 적용해 볼 수 있습니다.

예제: 렌탈 날짜를 'YYYY-MM-DD' 형식으로 변환

다음 SQL 쿼리는 렌탈이 일어난 날짜를 'YYYY-MM-DD' 형식으로 출력합니다.

```
SELECT
    rental_id,
    DATE_FORMAT(rental_date, '%Y-%m-%d') AS formatted_rental_date
FROM
    rental
LIMIT 5;
```

이 쿼리를 실행하면 아래와 같은 결과를 볼 수 있을 것입니다:

```
| rental_id | formatted_rental_date |
|-----------|-----------------------|
| 1         | 2005-05-24            |
| 2         | 2005-05-24            |
| 3         | 2005-05-24            |
| 4         | 2005-05-24            |
| 5         | 2005-05-24            |
```

여기서 `formatted_rental_date` 컬럼은 원래 `rental_date` 컬럼을 `DATE_FORMAT()` 함수를 사용하여 'YYYY-MM-DD' 형식으로 변환한 결과입니다.

3. 숫자 함수

`ABS(number)` : 숫자의 절대값을 반환합니다.

- 예: 아래 예제는 해당 쿼리에서 실제로 음수 값이 없음을 가정하고 작성된 것입니다.

```sql
SELECT ABS(-payment.amount) AS absolute_amount
FROM payment
LIMIT 10;
```

`CEIL(number)` : 숫자보다 크거나 같은 가장 작은 정수 값을 반환합니다.

- 예: `payment` 테이블에서 `amount` 컬럼의 값을 올림하여 조회합니다.

```sql
SELECT CEIL(amount) AS ceiling_amount
FROM payment
LIMIT 10;
```

`FLOOR(number)` : 숫자 이하의 가장 큰 정수 값을 반환합니다.

- 예: `payment` 테이블에서 `amount` 컬럼의 값을 내림하여 조회합니다.

```sql
SELECT FLOOR(amount) AS floor_amount
FROM payment
LIMIT 10;
```

`ROUND(number, decimals)` : 숫자를 특정 소수점 자리수로 반올림합니다.

- 예: `payment` 테이블에서 `amount` 컬럼의 값을 소수점 두 자리로 반올림하여 조회합니다.

```sql
SELECT ROUND(amount, 2) AS rounded_amount
FROM payment
LIMIT 10;
```

`SQRT(number)` : 숫자의 제곱근을 반환합니다.

- 예: `film` 테이블에서 `length` 컬럼의 제곱근을 계산합니다.

```
SELECT SQRT(length) AS sqrt_length
FROM film
LIMIT 10;
```

기본 연습문제

1. `film` 테이블에서 영화 제목(`title`)의 길이가 15자인 영화들을 찾아주세요.
2. `actor` 테이블에서 첫 번째 이름(`first_name`)이 소문자로 'john'인 배우들의 전체 이름을 대문자로 출력해주세요.
3. `film` 테이블에서 `description` 의 3번째 글자부터 6글자가 'Action'인 영화의 제목을 찾아주세요.
4. `rental` 테이블에서 대여 시작 날짜(`rental_date`)가 2006년 1월 1일 이후인 모든 대여에 대해, 예상 반납 날짜를 대여 날짜로부터 5일 뒤로 설정하여, 출력해주세요.
5. `payment` 테이블에서 결제 금액(`amount`)이 5 이하인 모든 결제에 대해, 절대값을 계산하여 출력해주세요.
6. `film` 테이블에서 영화 길이(`length`)가 120분 이상인 모든 영화에 대해, 영화 길이의 제곱근을 계산해주세요.
7. `payment` 테이블에서 결제 금액(`amount`)을 소수점 첫 번째 자리에서 반올림하여 출력해주세요.

서브 쿼리 중급

서브쿼리(Sub query)는 중첩 쿼리(Nested query) 라고도 하며, 다른 SQL 쿼리의 맥락 안에 포함되는 쿼리입니다. 추가적인 서브쿼리 예제를 테스트해보며, 서브 쿼리에 대해서도 보다 익숙해지도록 합니다.

다양하고 복잡한 서브쿼리 예제

1. 간단한 서브쿼리

평균 결제 금액보다 많은 결제를 한 고객을 찾아봅시다:

```
SELECT first_name, last_name
FROM customer
WHERE customer_id IN (
  SELECT customer_id
  FROM payment
  WHERE amount > (SELECT AVG(amount) FROM payment)
);
```

이 쿼리에서 가장 안쪽 쿼리는 평균 결제 금액을 계산합니다. 중간 쿼리는 이 평균 금액보다 큰 결제의 `customer_id` 를 찾습니다. 가장 바깥 쿼리는 이들 고객의 이름을 가져옵니다.

2. GROUP BY가 있는 서브쿼리

평균 결제 횟수보다 많은 결제를 한 고객을 찾아봅시다:

```
SELECT first_name, last_name
FROM customer
WHERE customer_id IN (
  SELECT customer_id
  FROM payment
  GROUP BY customer_id
  HAVING COUNT(*) > (
    SELECT AVG(payment_count)
    FROM (
      SELECT COUNT(*) AS payment_count
      FROM payment
      GROUP BY customer_id
    ) AS payment_counts
  )
);
```

이 쿼리에서 가장 안쪽 쿼리는 각 고객에 대한 결제 횟수를 세고 그 평균을 계산합니다. 중간 쿼리는 이 평균보다 더 많은 결제를 한 고객의 `customer_id` 를 찾습니다. 가장 바깥 쿼리는 이들 고객의 이름을 가져옵니다.

3. 최대값을 가진 행 찾기

가장 많은 결제를 한 고객을 찾아봅시다:

```
SELECT first_name, last_name
FROM customer
WHERE customer_id = (
  SELECT customer_id
  FROM (
    SELECT customer_id, COUNT(*) AS payment_count
    FROM payment
    GROUP BY customer_id
  ) AS payment_counts
  ORDER BY payment_count DESC
  LIMIT 1
);
```

이 쿼리에서는 가장 안쪽 쿼리가 각 고객의 결제 횟수를 계산하고, 중간 쿼리가 결제 횟수를 내림차순으로 정렬하여 가장 많은 결제를 한 고객의 `customer_id`를 찾습니다. 그리고 가장 바깥 쿼리가 해당 고객의 이름을 가져옵니다.

4. 상관 서브쿼리

각 고객에 대해 자신이 결제한 평균 금액보다 큰 결제를 한 경우의 결제 정보를 찾아봅시다:

```
SELECT P.customer_id, P.amount, P.payment_date
FROM payment P
WHERE P.amount > (
  SELECT AVG(amount)
  FROM payment
  WHERE customer_id = P.customer_id
);
```

이 쿼리에서는 각 결제에 대해 해당 결제를 한 고객의 평균 결제 금액을 계산하고, 그 평균보다 큰 결제를 찾습니다. 이런 종류의 서브쿼리를 상관 서브쿼리(Correlated Subquery)라고 합니다. 왜냐하면 서브쿼리가 외부 쿼리의 변수를 참조하기 때문입니다.

기본 연습문제

1. `film` 테이블에서 평균 영화 길이(`length`)보다 긴 영화들의 제목(`title`)을 찾아주세요.
2. `rental` 테이블에서 고객별 평균 대여 횟수보다 많은 대여를 한 고객들의 이름(`first_name`, `last_name`)을 찾아주세요.
3. 가장 많은 영화를 대여한 고객의 이름(`first_name`, `last_name`)을 찾아주세요.
4. 각 고객에 대해 자신이 대여한 평균 영화 길이(`length`)보다 긴 영화들의 제목(`title`)을 찾아주세요.

복합 연습문제

1. `rental`과 `inventory` 테이블을 JOIN하고, `film` 테이블에 있는 `replacement_cost`가 $20 이상인 영화를 대여한 고객의 이름을 찾으세요. 고객 이름은 소문자로 출력해주세요.
2. `film` 테이블에서 `rating`이 'PG-13'인 영화들에서, `description`의 길이가, `rating`이 'PG-13'인 영화들의 평균 description 길이보다 긴 영화의 제목을 찾으세요.
3. `customer`와 `rental`, `inventory`, `film` 테이블을 JOIN하여, 2005년 8월에 대여된 모든 'R' 등급 영화의 제목과 해당 영화를 대여한 고객의 이메일을 찾으세요.
4. `payment` 테이블에서 최근 30일 이내의 모든 결제를 찾고, 결제 금액을 소수점 둘째 자리에서 반올림하여 출력하세요. 이때, 각 고객별 총 결제 금액과 평균 결제 금액도 함께 출력해주세요.

5. `actor` 와 `film_actor` , `film` 테이블을 JOIN하고, 'Science Fiction' 카테고리에 속한 영화에 출연한 배우의 이름을 찾으세요. 배우의 이름은 성과 이름을 연결하여 대문자로 출력해주세요.

UNION, UNION ALL, INTERSECT, 그리고 EXCEPT

두 개 이상의 SELECT 문의 결과를 결합하거나 비교하도록 해주는 집합 연산입니다.

1. `UNION` : 두 개 이상의 SELECT 문의 결과 집합을 결합하며, 중복된 행은 제거합니다. 각 SELECT 문의 열은 같은 순서를 가져야 하며, 유사한 데이터 유형을 가지고 있어야 합니다.
2. `UNION ALL` : 두 개 이상의 SELECT 문의 결과 집합을 결합하며, 중복된 행도 포함합니다. UNION과 마찬가지로, 각 SELECT 문의 열은 같은 순서를 가져야 하며, 유사한 데이터 유형을 가지고 있어야 합니다.
3. `INTERSECT` : 두 개 이상의 SELECT 문의 결과 집합의 교집합을 반환합니다. 즉, 모든 SELECT 문에 공통적으로 있는 행을 반환합니다. UNION과 UNION ALL과 마찬가지로, 각 SELECT 문의 열은 같은 순서를 가져야 하며, 유사한 데이터 유형을 가지고 있어야 합니다.
4. `EXCEPT` : 두 SELECT 문의 결과 집합의 차집합을 반환합니다. 즉, 첫 번째 결과 집합에는 있지만 두 번째 결과 집합에는 없는 행을 반환합니다. 각 SELECT 문의 열은 같은 순서를 가져야 하며, 유사한 데이터 유형을 가지고 있어야 합니다.

1. UNION

`film` 테이블이나 `inventory` 테이블에 있는 영화 ID 리스트를 가져오려면 다음과 같이 쿼리를 작성할 수 있습니다:

```
SELECT film_id FROM film
UNION
SELECT film_id FROM inventory;
```

이 쿼리는 `film` 또는 `inventory` 테이블에 있는 고유한 `film_id` 리스트를 반환합니다.

2. UNION ALL

결과에 중복 `film_id` 를 포함시키려면 `UNION ALL` 을 사용할 수 있습니다:

```
SELECT film_id FROM film
UNION ALL
SELECT film_id FROM inventory;
```

이 쿼리는 `film` 및 `inventory` 테이블의 모든 `film_id` 리스트를 반환하며, 중복 값도 포함합니다.

3. INTERSECT

만약 `film` 테이블과 `inventory` 테이블 모두에 있는 `film_id` 리스트만 가져오려면 `INTERSECT` 를 사용할 수 있습니다:

```
SELECT film_id FROM film
INTERSECT
SELECT film_id FROM inventory;
```

이 쿼리는 `film` 및 `inventory` 테이블 모두에 있는 `film_id` 리스트를 반환합니다.

4. EXCEPT

`film` 테이블에는 있지만 `inventory` 테이블에는 없는 `film_id` 리스트를 가져오려면 `EXCEPT` 를 사용할 수 있습니다:

```
SELECT film_id FROM film
EXCEPT
SELECT film_id FROM inventory;
```

이 쿼리는 `film` 테이블에는 있지만 `inventory` 테이블에는 없는 `film_id` 리스트를 반환합니다.

> 집합 연산을 사용할 때는 SELECT 문이 동일한 수의 열을 선택해야 하며, 해당 열은 호환 가능한 데이터 유형을 가져야 하고, 열은 동일한 순서로 있어야 한다는 점을 기억하세요. 또한 모든 데이터베이스 시스템이 `INTERSECT` 와 `EXCEPT` 를 지원하는 것은 아니라는 점도 참고하세요.

연습문제

1. `film` 테이블과 `film_category` 테이블에서 각각 중복 없이 `film_id` 를 조회하는 SQL문을 작성해 보세요.
2. `film` 테이블과 `film_category` 테이블에서 각각 `film_id` 를 조회하되 중복값도 포함하는 SQL문을 작성해 보세요.
3. `film` 테이블과 `film_category` 테이블에서 모두 나오는 `film_id` 를 조회하는 SQL문을 작성해 보세요.
4. `film` 테이블에는 존재하지만 `film_category` 테이블에는 존재하지 않는 `film_id` 를 조회하는 SQL문을 작성해 보세요.

트랜잭션, COMMIT 그리고 ROLLBACK

트랜잭션은 하나 이상의 SQL 문을 포함하는 작업의 논리적 단위입니다. 트랜잭션은 단일 논리적 작업 단위로 수행되는 연산의 순서입니다. 논리적 작업 단위는 트랜잭션으로 간주되기 위해 원자성, 일관성, 고립성, 그리고 지속성 (ACID)의 네 가지 특성을 가져야 합니다.

1. `원자성(Atomicity)` : 이 특성은 트랜잭션이 불가분의 단위로 처리되어야 함을 나타냅니다. 이는 트랜잭션의 모든 연산이 실행되거나 아무것도 실행되지 않아야 함을 의미합니다. 트랜잭션의 일부가 실패하면 전체 트랜잭션이 실패하며, 데이터베이스 상태는 변경되지 않습니다.
2. `일관성(Consistency)` : 일관성 특성은 트랜잭션이 데이터베이스를 하나의 유효한 상태에서 다른 유효한 상태로 유지하도록 보장하며, 이는 트랜잭션이 유효한 결과를 가져야 함을 의미합니다.
3. `고립성(Isolation)` : 고립성 특성은 트랜잭션의 동시 실행이 데이터베이스를 마치 트랜잭션이 순차적으로 실행된 것처럼 같은 상태로 남겨놓도록 보장합니다.
4. `지속성(Durability)` : 지속성은 트랜잭션이 일단 커밋되면 시스템 실패가 발생하더라도 커밋 상태가 유지되도록 보장합니다.

`COMMIT` 과 `ROLLBACK` 은 트랜잭션을 제어하는 SQL 명령입니다.

1. `COMMIT` : 이 명령은 현재 트랜잭션에서 만든 모든 변경 사항을 저장하는 데 사용됩니다. COMMIT 문 바로 다음에 새 트랜잭션이 시작됩니다.
2. `ROLLBACK` : 이 명령은 현재 트랜잭션에서 만든 일부 또는 모든 변경 사항을 취소합니다. 또한 현재 트랜잭션을 종료하며, 새로운 트랜잭션이 시작됩니다.

예제

트랜잭션을 지원하는 데이터베이스를 사용한다고 가정하고, 다음 예제를 고려해봅시다. `orders` 라는 테이블이 있고, 이 테이블에는 `order_id` , `product_id` , `quantity` , `price` 그리고 `status` 라는 열이 있다고 가정합시다:

```
START TRANSACTION;

UPDATE orders
SET status = 'Processed'
WHERE order_id = 101;

UPDATE orders
SET status = 'Processed'
WHERE order_id = 102;

COMMIT;
```

이 예제에서는 order_id가 101과 102인 주문의 상태를 'Processed'로 설정합니다. `START TRANSACTION;` 문은 트랜잭션의 시작을 표시합니다. `COMMIT;` 문은 마지막 `COMMIT` 또는 `ROLLBACK` 명령 이후에 만든 모든 수정 사항을 저장합니다.

이제 `ROLLBACK` 을 사용하는 예제를 고려해봅시다:

```
START TRANSACTION;

UPDATE rental
SET return_date = NOW()
WHERE rental_id = 3;

UPDATE rental
SET return_date = NOW()
WHERE rental_id = 4;

ROLLBACK;
```

이 예제에서는 `ROLLBACK;` 문이 `orders` 테이블에 대한 업데이트를 취소합니다. 두 개의 `UPDATE` 문은 실행되지만, `ROLLBACK` 문으로 인해 그 효과는 데이터베이스에 저장되지 않습니다. 따라서 order_id가 103과 104인 주문의 상태는 데이터베이스에서 변경되지 않습니다.

> 트랜잭션은 특히 여러 트랜잭션이 동시에 실행되는 데이터베이스 시스템에서 데이터 무결성을 유지하는 데 중요합니다.

연습문제

1. 'payment' 테이블에서 payment_id가 1001인 amount 를 3.99 로 변경하는 트랜잭션을 시작하고, 그 트랜잭션을 커밋하는 SQL문을 작성해 보세요.
2. 'payment' 테이블에서 payment_id가 1001인 amount 를 2.99 로 변경하는 트랜잭션을 시작하고, 그 트랜잭션을 커밋하는 SQL문을 작성해 보세요.

SQL VIEW 사용법

VIEW란 무엇인가?

VIEW는 실제 테이블을 기반으로 한 가상 테이블입니다. VIEW를 사용하면 복잡한 쿼리를 단순화하고, 데이터의 특정 부분에만 접근을 허용할 수 있습니다.

VIEW 생성하기

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

예제:

```
CREATE VIEW ActorInfo AS
SELECT first_name, last_name
FROM actor
WHERE actor_id < 100;
```

VIEW 수정하기

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

VIEW 삭제하기

```
DROP VIEW view_name;
```

연습문제

1. ActorInfo라는 VIEW를 만드세요. 이 VIEW는 `actor` 테이블에서 `first_name` 과 `last_name` 컬럼을 포함해야 합니다. `actor_id` 가 50 미만인 배우만 포함해야 합니다.
2. `film` 테이블에서 렌탈 비용이 $2.00보다 높은 영화에 대한 VIEW를 만들어 봅니다. 이 VIEW의 이름은 `ExpensiveFilms` 이고, `title` 과 `rental_rate` 컬럼만을 포함해야 합니다.
3. 이미 만든 VIEW인 `ActorInfo` 를 수정하여 `actor_id` 가 100 미만인 배우만 포함하도록 만들어 봅니다.
4. `ExpensiveFilms` VIEW를 삭제합니다.

빈도는 적지만, 가볍게는 알아둬야할 고급 SQL

WITH 절 (Common Table Expressions)

`WITH` 절은 Common Table Expressions (CTEs)라고도 불리며, 일시적인 결과 세트를 만들어 `SELECT`, `INSERT`, `UPDATE`, `DELETE` 문에서 참조할 수 있게 해줍니다. CTE 는 단일 SQL 쿼리 내에서만 사용 가능합니다. (즉 SQL 쿼리가 종료하면 자동 삭제됩니다.) WITH 절은 복잡한 쿼리를 쉽게 분해할 수 있다는 장점을 가지고 있습니다.

문법

```
WITH cte_name AS (
    -- SQL 쿼리
)
-- CTE를 사용하는 메인 쿼리
```

예시

예를 들어, 재고에도 있는 모든 영화의 이름을 가져오고 싶다면 아래와 같이 작성할 수 있습니다.

```
WITH FilmInventory AS (
    SELECT DISTINCT film_id FROM inventory
)
SELECT f.film_id, f.title
FROM film f
JOIN FilmInventory fi ON f.film_id = fi.film_id;
```

CASE WHEN 절

`CASE WHEN` 절을 사용하면 SQL 쿼리에 조건 로직을 추가할 수 있습니다. 프로그래밍에서 if-else 문과 비슷한 역할을 합니다.

문법

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE result
END
```

예시

`sakila` 데이터베이스의 영화를 렌탈 비용을 기준으로 카테고리화하고 싶다면 이렇게 쓸 수 있습니다.

```
SELECT title,
CASE
    WHEN rental_rate < 1 THEN 'Cheap'
    WHEN rental_rate BETWEEN 1 AND 3 THEN 'Moderate'
    ELSE 'Expensive'
END AS PriceCategory
FROM film;
```

연습문제

문제 1: WITH 를 사용해서, `sakila` 데이터베이스의 각 등급별 영화의 평균 길이를 알아보세요.

문제 2: `CASE WHEN` 을 사용해서 `customer` 테이블의 고객들을 `active` 컬럼에 따라 'Active' 또는 'Inactive'로 분류해보세요.

문제 3: WITH 를 사용해서, `sakila` 의 `film` 테이블에서 각 `rating` 에 따른 평균 `rental_duration` 을 계산해보세요.

문제 4: `sakila` 의 `customer` 테이블에서 `active` 컬럼에 따라 고객을 'Active' 또는 'Inactive'로 분류하고, 각 분류에 몇 명이 있는지 계산하세요.

문제 5: WITH 도 사용해서, `sakila` 의 `payment` 테이블에서 각 고객별 총 지불액을 계산하고, 그 지불액에 따라 고객을 'Low', 'Medium', 'High'로 분류하세요. 분류 기준은 다음과 같습니다:

- Low: $0-50$

- Medium: 51−100
- High: $100 이상

GROUP_CONCAT() 기본 문법

`GROUP_CONCAT()` 은 MySQL에서 제공하는 문자열 집계 함수로, 그룹 내의 여러 행을 하나의 문자열로 결합합니다. 기본 문법은 다음과 같습니다.

```
GROUP_CONCAT(expression)
GROUP_CONCAT(expression SEPARATOR separator_string)
```

- `expression` : 결합할 컬럼 또는 표현식
- `separator_string` : 행을 연결할 때 사용할 문자열 (기본값은 쉼표 `,`)

예를 들어, 여러 사원들이 다른 부서에 속해 있다고 할 때, 각 부서별로 속한 사원의 이름을 하나의 문자열로 만들고 싶다면 `GROUP_CONCAT()` 을 사용할 수 있습니다. `GROUP_CONCAT()` 함수는 여러 행의 데이터를 하나의 문자열로 표현할 때 매우 유용합니다.

sakila 데이터베이스를 이용한 간단한 예제

`sakila` 데이터베이스에서는 영화 대여 정보를 다루고 있습니다. 이를 기반으로 `GROUP_CONCAT()` 을 사용하는 간단한 예제를 살펴보겠습니다.

예제: 고객별로 렌탈한 영화의 제목을 문자열로 만들기

다음의 SQL 쿼리는 고객 ID, 고객의 이름과 함께 그 고객이 렌탈한 영화 제목을 하나의 문자열로 반환합니다.

```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    GROUP_CONCAT(f.title ORDER BY f.title ASC) AS rented_movies
FROM
    customer c
JOIN
    rental r ON c.customer_id = r.customer_id
JOIN
    inventory i ON r.inventory_id = i.inventory_id
JOIN
    film f ON i.film_id = f.film_id
GROUP BY
    c.customer_id
LIMIT 5;
```

이 쿼리를 실행하면, 각 고객이 렌탈한 영화 제목이 알파벳 순으로 정렬된 문자열로 나타납니다. 결과는 아래와 같을 것입니다.

```
| customer_id | customer_name | rented_movies                          |
|-------------|---------------|----------------------------------------|
| 1           | Mary Smith    | Movie1, Movie2, Movie3                 |
| 2           | John Doe      | Movie1, Movie4, Movie5                 |
| 3           | Emily Davis   | Movie2, Movie3, Movie6                 |
| ...         | ...           | ...                                    |
```