7. 뷰(View)

#0.강의/2.데이터베이스로드맵/2.기본

- /뷰(View) 소개
- /뷰 생성, 조회, 수정, 삭제
- /뷰의 장점과 단점
- /문제와 풀이
- /정리

뷰(View) 소개

우리는 지난 시간에 `JOIN`과 `CASE` 문, `GROUP BY`를 총동원하여 '카테고리별, 상태별 주문 현황'이라는 매우 유용한 보고서를 만드는 쿼리를 완성했다.

```
-- 지난 시간에 만든 복잡하고 유용한 쿼리
SELECT
    p.category,
    COUNT(*) AS total_orders,
    SUM(CASE WHEN o.status = 'COMPLETED' THEN 1 ELSE 0 END) AS
completed_count,
    SUM(CASE WHEN o.status = 'SHIPPED' THEN 1 ELSE 0 END) AS shipped_count,
    SUM(CASE WHEN o.status = 'PENDING' THEN 1 ELSE 0 END) AS pending_count
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
GROUP BY
    p.category;
```

이 쿼리는 매우 유용해서, 우리 쇼핑몰의 재무팀, 마케팅팀, 운영팀에서 매일 아침 확인해야 하는 핵심 지표라고 가정해 보자. 여기서 오늘의 문제 상황이 발생한다.

- **복잡성**: 이 쿼리는 너무 길고 복잡하다. SQL에 익숙하지 않은 직원이 이 쿼리를 매번 실수 없이 정확하게 입력하기란 거의 불가능에 가깝다.
- **재사용성**: 여러 팀의 여러 사람이 이 쿼리를 사용하려면, 각자 이 긴 쿼리문을 어딘가에 저장해두고 복사-붙여넣기

를 해야 한다. 만약 보고서의 로직이 변경되면(예: '반품 완료' 상태 추가), 모든 사람이 각자 저장해 둔 쿼리를 수정해야 하는 끔찍한 상황이 발생한다.

- **보안**: 이 쿼리를 실행하려면, 사용자는 원본 테이블인 `orders`와 `products`에 직접 접근할 수 있는 권한(`SELECT` 권한)이 있어야 한다. 하지만 운영팀 직원에게는 고객의 개인정보나 상품의 원가 같은 민감한 정보가 담겨있을 수 있는 원본 테이블 전체를 보여주고 싶지 않다. 딱 이 요약된 보고서 내용만 보게 하고 싶다.

왜 우리는 이런 불편함과 위험을 감수해야만 할까? 이 복잡한 쿼리 자체를 데이터베이스에 하나의 '바로 가기'처럼 저장해두고, 필요할 때마다 간단한 이름으로 불러 쓸 수는 없을까?

이 모든 문제를 한 번에 해결해 주는 마법 같은 도구가 바로 **뷰(VIEW)**다.

뷰(VIEW)의 개념

뷰는 아주 간단하게 정의할 수 있다.
뷰(View)는 실제 데이터를 가지고 있지 않은 '가상의 테이블'이다. 그 실체는 데이터베이스에 이름과 함께 저장된 하나의 `SELECT` 쿼리문이다.

컴퓨터 바탕화면의 '바로 가기(Shortcut)' 아이콘을 생각하면 쉽다.

- 바로 가기 아이콘은 그 자체로 프로그램 파일이 아니다. 단지 실제 프로그램이 어디에 있는지 위치 정보만 담고 있다.
- 우리는 복잡한 경로를 찾아 들어갈 필요 없이, 바로 가기 아이콘을 더블클릭하는 것만으로 프로그램을 실행할 수 있다.

뷰도 똑같다.

- 뷰는 그 자체로 데이터를 저장하는 테이블이 아니다. 단지 복잡한 `SELECT` 쿼리문 자체를 저장하고 있다.
- 우리는 복잡한 쿼리를 실행할 필요 없이, `SELECT * FROM 나의_바로가기_뷰;` 라는 간단한 명령만으로 그 복잡한 쿼리의 결과를 얻을 수 있다.

뷰(VIEW)의 동작 원리

사용자가 `v_category_order_status` 라는 뷰를 조회하면, 데이터베이스 내부에서는 다음과 같은 일이 일어난다.

1. 사용자가 `SELECT * FROM v_category_order_status;` 라는 쿼리를 실행한다.
2. 데이터베이스는 `v_category_order_status`라는 이름의 실제 테이블을 찾지 않는다. 대신, 이 뷰에 정의된 (저장된) 원래의 긴 `SELECT` 쿼리문을 찾아낸다.
3. 데이터베이스는 그 저장된 `SELECT` 쿼리문(우리가 위에서 만든 복잡한 쿼리)을 **그 순간에 실행**한다.
4. 실행 결과가 사용자에게 반환된다. 사용자는 마치 그냥 테이블을 조회한 것처럼 느끼게 된다.

여기서 중요한 점은, 뷰는 데이터를 저장하지 않기 때문에, 우리가 뷰를 조회할 때마다 항상 최신 상태의 원본 테이블(`orders`, `products`)을 기준으로 쿼리가 실행된다는 것이다. 따라서 **뷰의 데이터는 항상 최신 상태를 유지**한다.

뷰(VIEW)를 사용하는 이유

1. **편리성**: 복잡한 쿼리를 단순화하여 쉽게 재사용할 수 있다.
2. **보안성**: 사용자에게 원본 테이블에 대한 접근 권한을 주지 않고, 뷰를 통해서만 제한된 데이터(특정 행이나 특정 컬럼)에 접근하도록 허용할 수 있다. 민감한 정보를 숨기는 데 매우 효과적이다.
3. **논리적 독립성**: 만약 나중에 원본 테이블의 구조가 일부 변경되더라도, 뷰의 정의만 수정하면 뷰를 사용하는 응용 프로그램은 코드를 바꿀 필요가 없을 수도 있다. 뷰가 중간에서 변화를 흡수하는 '추상화 계층' 역할을 하기 때문이다.

뷰는 단순히 편의 기능이 아니라, 데이터베이스의 보안과 유지보수성을 크게 향상시키는 실무의 핵심 기능이다.

이제 뷰가 왜 필요한지 충분히 이해했을 것이다. 다음 시간에는 이토록 유용한 뷰를 실제로 어떻게 만들고(`CREATE`), 조회하고, 수정하고, 삭제하는지에 대한 구체적인 문법을 알아보겠다.

뷰 생성, 조회, 수정, 삭제

이번 시간에는 뷰를 직접 구현해보자. 지난 시간에 사용했던 '카테고리별 주문 현황' 쿼리를 `v_category_order_status` 라는 이름의 뷰로 직접 생성하고, 일반 테이블처럼 다루는 전 과정을 실습해 보자.

`CREATE VIEW`: 뷰 생성하기

뷰를 생성하는 명령어는 `CREATE VIEW`이다. 문법은 아주 직관적이다.

```
CREATE VIEW 뷰이름 AS SELECT 쿼리문;
```

이제 이 문법에 따라, 우리의 복잡한 쿼리를 `v_category_order_status` 라는 이름의 뷰로 데이터베이스에 저장해 보자. 뷰 이름 앞에는 `v_` 나 `view_` 같은 접두사를 붙여서 다른 테이블과 쉽게 구분할 수 있도록 하는 것이 실무에

서 흔히 사용하는 좋은 습관이다.

```
DROP VIEW IF EXISTS v_category_order_status; -- 만약 뷰가 이미 존재한다면 제거

CREATE VIEW v_category_order_status AS
SELECT
    p.category,
    COUNT(*) AS total_orders,
    SUM(CASE WHEN o.status = 'COMPLETED' THEN 1 ELSE 0 END) AS
completed_count,
    SUM(CASE WHEN o.status = 'SHIPPED' THEN 1 ELSE 0 END) AS shipped_count,
    SUM(CASE WHEN o.status = 'PENDING' THEN 1 ELSE 0 END) AS pending_count
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
GROUP BY
    p.category;
```

이 쿼리를 실행하고 나면, 우리 데이터베이스에 `v_category_order_status` 라는 새로운 객체가 생성된다. 이 뷰는 실제 데이터를 담고 있는 것이 아니라, 위 `SELECT` 쿼리문 자체를 품고 있는 것이다.

뷰 조회하기: `SELECT`

뷰를 조회하는 것은 너무나도 간단하다. 뷰를 그냥 하나의 테이블이라고 생각하고 `SELECT` 문을 사용하면 된다.
이제 저 길고 복잡한 쿼리 대신, 우리가 만든 뷰의 이름을 사용하여 결과를 불러와 보자.

```
SELECT * FROM v_category_order_status;
```

[실행 결과]

category	total_orders	completed_count	shipped_count	pending_count
전자기기	5	2	2	1
도서	2	2	0	0

간단한 쿼리 한 줄로 복잡한 쿼리를 실행했을 때와 완벽하게 동일한 결과를 얻었다.

뷰 역시 테이블처럼 다룰 수 있으므로, `WHERE` 절이나 `ORDER BY`를 사용해서 뷰의 결과를 필터링하거나 정렬할 수도 있다.

```
SELECT *
FROM v_category_order_status
WHERE category = '전자기기';
```

[실행 결과]

category	total_orders	completed_count	shipped_count	pending_count
전자기기	5	2	2	1

`ALTER VIEW`: 뷰 수정하기

뷰를 사용하다 보면, 저장된 쿼리의 내용을 수정해야 할 일이 생긴다. 예를 들어, 마케팅팀에서 "카테고리별 주문 건수뿐만 아니라, 총 매출액도 함께 보고 싶어요." 라는 추가 요청을 했다고 가정해 보자.

이때 사용하는 명령어가 `ALTER VIEW`이다. `CREATE VIEW`와 문법은 거의 동일하며, 기존 뷰의 정의를 새로운 `SELECT` 쿼리로 덮어쓴다.

```
ALTER VIEW v_category_order_status AS
SELECT
    p.category,
    SUM(p.price * o.quantity) AS total_sales, -- 매출액 컬럼 추가!
    COUNT(*) AS total_orders,
    SUM(CASE WHEN o.status = 'COMPLETED' THEN 1 ELSE 0 END) AS
completed_count,
    SUM(CASE WHEN o.status = 'SHIPPED' THEN 1 ELSE 0 END) AS shipped_count,
    SUM(CASE WHEN o.status = 'PENDING' THEN 1 ELSE 0 END) AS pending_count
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
```

```
GROUP BY
    p.category;
```

뷰의 정의를 수정한 뒤, 다시 한번 뷰를 조회해 보자.

```
SELECT * FROM v_category_order_status;
```

[실행 결과]

category	total_sales	total_orders	completed_count	shipped_count	pending_ct
전자기기	770000	5	2	2	1
도서	84000	2	2	0	0

`total_sales`라는 새로운 컬럼이 성공적으로 추가된 것을 확인할 수 있다. 뷰를 사용하는 모든 팀은 이제 별도의 수정 없이 이 새로운 보고서를 즉시 받아볼 수 있게 된다.

`DROP VIEW`: 뷰 삭제하기

더 이상 사용하지 않는 뷰는 `DROP VIEW` 명령어로 간단하게 삭제할 수 있다.

```
DROP VIEW v_category_order_status;
```

이 명령어를 실행하면 뷰의 정의, 즉 우리가 저장했던 '바로 가기'가 데이터베이스에서 사라진다.

여기서 가장 중요한 사실은, **뷰를 삭제해도 원본 테이블인 `orders`와 `products`의 데이터는 단 하나도 손상되거나 변경되지 않는다**는 점이다. 그저 쿼리를 저장해 둔 편리한 '도구' 하나가 사라지는 것뿐이다.

지금까지 뷰를 생성하고, 활용하고, 관리하는 기본적인 방법을 배웠다. 뷰의 편리함은 충분히 느꼈을 것이다. 하지만 실무에서 어떤 기술을 사용하기 전에는, 그 기술의 장점뿐만 아니라 단점과 주의사항까지 명확히 알아야 한다.

다음 시간에는 뷰의 장단점을 실무적인 관점에서 깊이 있게 파고들어, 언제 뷰를 써야 하고 언제 쓰지 말아야 하는지에

알아보겠다.

뷰의 장점과 단점

지금까지 우리는 뷰를 만들고, 조회하고, 수정하고, 삭제하는 방법을 배우며 그 편리함을 맛보았다. 복잡한 쿼리가 단 한 줄로 줄어드는 마법을 보며 '이거 정말 좋은데?' 라고 생각했을 것이다.

물론 뷰는 매우 훌륭한 도구다. 하지만 실무에서 어떤 기술을 제대로 사용하려면, 그 기술이 가진 빛과 그림자를 모두 알아야 한다. 즉, 장점뿐만 아니라 단점과 주의사항까지 명확히 이해해야만 오남용을 막고 적재적소에 활용할 수 있다.

이번 시간에는 뷰의 장단점을 실무적인 관점에서 정리해 보겠다.

뷰의 장점

1. 편리성과 재사용성

가장 큰 장점이다. 복잡한 `SELECT` 쿼리(수십 줄에 달하는 `JOIN`, 서브쿼리, `CASE` 문의 조합)를 뷰 뒤에 숨겨둘 수 있다.

- **사용자 입장**: 쿼리의 내부 로직을 전혀 몰라도, `SELECT * FROM v_my_report;` 한 줄만으로 원하는 결과를 얻을 수 있다.
- **개발자 입장**: 동일한 로직이 여러 곳에서 필요할 때, 뷰 하나만 만들어두면 모두가 재사용할 수 있다. 만약 로직을 수정해야 할 경우, 뷰의 정의(`ALTER VIEW`)만 한 번 수정하면 이 뷰를 사용하는 모든 곳에 즉시 반영되므로 유지보수성이 극적으로 향상된다.

2. 보안성

실무에서 뷰를 사용하는 가장 중요한 이유 중 하나다. 뷰는 데이터베이스에 대한 섬세한 '권한 제어'를 가능하게 한다.

- **특정 컬럼 숨기기**: `employees` 테이블에서 `salary`(연봉)나 `private_info`(개인정보) 컬럼을 제외한 뷰를 만들어, 일반 직원들에게는 해당 뷰만 조회할 수 있는 권한을 줄 수 있다.
- **특정 행만 노출하기**: 마케팅팀 직원에게는 `orders` 테이블에서 '마케팅' 캠페인을 통해 들어온 주문 데이터만 보여주는 뷰를 만들어 제공할 수 있다.

이렇게 하면, 사용자에게 원본 테이블에 대한 직접적인 접근 권한을 주지 않고도, 그들이 꼭 봐야 할 데이터만 안전하게 노출하는 것이 가능해진다.

3. 논리적 데이터 독립성

뷰는 일종의 '추상화 계층' 역할을 한다.

만약 데이터베이스 구조를 변경해야 하는 상황(테이블 분리, 컬럼명 변경 등)이 발생했다고 하자. 이때, 변경 이전과 동일한 구조의 뷰를 새로 만들어 준다면, 이 뷰를 사용하던 응용 프로그램들은 원본 테이블이 변경되었다는 사실조차 모른 채 기존 코드를 그대로 사용할 수 있다. 뷰가 중간에서 물리적인 테이블의 변화를 숨겨주는 방패 역할을 하는 셈이다.

뷰의 단점 및 주의사항

이제 그림자, 즉 뷰를 사용할 때 반드시 주의해야 할 점들을 알아보자.

1. 성능 문제

`SELECT * FROM my_view;` 라는 간단한 쿼리 뒤에는 어마어마하게 무거운 실제 쿼리가 숨어있을 수 있다.

- **착각**: 사용자는 자신이 매우 가벼운 쿼리를 실행한다고 착각하지만, 실제로는 시스템에 큰 부하를 주는 쿼리를 날리고 있을 수 있다.
- **최적화의 한계**: 데이터베이스의 쿼리 옵티마이저는 매우 똑똑하지만, 뷰의 구조가 너무 복잡해지면 최적의 실행 계획을 찾아내는 데 어려움을 겪을 수 있다. 특히 **뷰 위에 또 다른 뷰를 쌓는 방식(Nested Views)**은 성능 저하의 주된 원인이 되므로 가급적 피해야 한다.

2. 업데이트 제약

뷰를 테이블처럼 다룰 수 있다고 했지만, 그것은 주로 **조회(읽기)**에 국한된 이야기다.

- **수정 불가(대부분의 경우)**: `JOIN`, 집계 함수(`SUM`, `COUNT` 등), `GROUP BY`, `DISTINCT` 등을 사용한 복잡한 뷰는 일반적으로 `INSERT`, `UPDATE`, `DELETE`가 불가능하다. 왜냐하면 데이터베이스 입장에서, 뷰의 한 행을 수정하는 것이 원본 테이블들의 데이터를 어떻게 바꿔야 하는지에 대한 명확한 규칙을 특정할 수 없기 때문이다.
- **수정 가능**: 뷰가 오직 하나의 기본 테이블만을 참조하고, 뷰의 모든 컬럼이 기본 테이블의 실제 컬럼을 직접 참조하는 경우에는 뷰에 데이터를 추가/수정할 수 있다. 이 경우 원본 테이블의 값이 변경된다.
- **실무 규칙**: **"뷰는 기본적으로 조회용이다"** 라고 생각하는 것이 가장 안전하고 일반적인 원칙이다. 데이터를 수정해야 한다면, 뷰가 아닌 원본 테이블에 직접 수행해야 한다.

정리하자면, 뷰는 만능 해결책이 아니다. 복잡한 로직을 편리하게 재사용하고 보안을 강화하는 데는 좋은 도구지만, 성능 저하의 가능성을 내포하고 있으며 데이터 수정에는 제약이 따른다는 한계를 가지고 있다.

지금까지 우리는 데이터를 조회하고, 가공하고, 합치고, 저장하는 등 SQL의 다양한 문법을 배웠다. 하지만 우리 쇼핑몰이 폭발적으로 성장해서 `users` 테이블에 수백만 명의 회원, `products` 테이블에 수십만 개의 상품 데이터가 쌓인다면 어떻게 될까?

```
SELECT * FROM users WHERE email = 'sean@example.com';
```

지금은 눈 깜짝할 사이에 실행되는 이 간단한 쿼리조차, 데이터가 많아지면 실행하는 데 수 초, 심지어는 수 분이 걸리는 끔찍한 상황이 발생할 수 있다. 왜 데이터가 많아지면 검색 속도가 느려지는 걸까?

다음 섹션에서는 데이터베이스 성능의 가장 핵심적인 비밀, 바로 **인덱스(INDEX)**의 세계로 떠나보겠다.

문제와 풀이

문제1: 고객의 기본 정보만 보여주는 뷰 생성하기

[문제]

`users` 테이블에는 고객의 주소, 생년월일과 같은 민감한 정보가 포함되어 있다. 마케팅팀에서는 고객에게 이메일을 보내기 위해 이름과 이메일 주소만 필요하다.

보안을 위해 고객의 `user_id`, `name`, `email` 컬럼만을 포함하는 `v_customer_email_list`라는 이름의 뷰(View)를 생성해라.

[실행 결과]

user_id	고객명	이메일
1	션	sean@example.com
2	네이트	nate@example.com
3	세종대왕	sejong@example.com
4	이순신	sunsin@example.com
5	마리 퀴리	marie@example.com
6	레오나르도 다빈치	vinci@example.com

[정답]

```sql
-- 기존에 뷰가 있다면 삭제
DROP VIEW IF EXISTS v_customer_email_list;

-- 뷰 생성
CREATE VIEW v_customer_email_list AS
SELECT
    user_id,
    name AS 고객명,
    email AS 이메일
FROM
    users;

-- 생성된 뷰 조회
SELECT * FROM v_customer_email_list;
```

문제2: 주문 상세 정보를 통합한 뷰 생성하기

[문제]

운영팀에서는 현재 주문 내역을 확인할 때마다 `orders`, `users`, `products` 테이블을 매번 `JOIN`해야 해서 불편함을 겪고 있다.

주문 ID(`order_id`), 고객 이름(`name`), 상품 이름(`name`), 주문 수량(`quantity`), 주문 상태(`status`)를 한 번에 볼 수 있는 `v_order_summary`라는 이름의 뷰를 생성하여 업무 효율을 높여라.

[실행 결과]

order_id	고객명	상품명	주문수량	주문상태
1	션	프리미엄 게이밍 마우스	1	COMPLETED
2	션	관계형 데이터베이스 입문	2	COMPLETED
3	네이트	기계식 키보드	1	SHIPPED
4	세종대왕	관계형 데이터베이스 입문	1	COMPLETED
5	이순신	4K UHD 모니터	1	PENDING

6	마리 퀴리	프리미엄 게이밍 마우스	1	COMPLETED
7	네이트	프리미엄 게이밍 마우스	2	SHIPPED

[정답]

```sql
-- 기존에 뷰가 있다면 삭제
DROP VIEW IF EXISTS v_order_summary;

-- 뷰 생성
CREATE VIEW v_order_summary AS
SELECT
    o.order_id,
    u.name AS '고객명',
    p.name AS '상품명',
    o.quantity AS '주문수량',
    o.status AS '주문상태'
FROM
    orders o
JOIN
    users u ON o.user_id = u.user_id
JOIN
    products p ON o.product_id = p.product_id;

-- 생성된 뷰 조회
SELECT * FROM v_order_summary;
```

문제3: '전자기기' 카테고리 매출 분석 뷰 생성하기

[문제]

전략기획팀은 '전자기기' 카테고리의 상품들에 대한 주문 건수와 총매출액을 집중적으로 분석하고자 한다.

`products` 테이블과 `orders` 테이블을 `JOIN`하여, '전자기기' 카테고리에 속한 상품들의 총주문 건수(`total_orders`)와 총매출액(`total_sales`)을 계산하는 `v_electronics_sales_status` 뷰를 생성해라.

[실행 결과]

category	total_orders	total_sales
전자기기	5	770000

[정답]

```sql
-- 기존에 뷰가 있다면 삭제
DROP VIEW IF EXISTS v_electronics_sales_status;

-- 뷰 생성
CREATE VIEW v_electronics_sales_status AS
SELECT
    p.category,
    COUNT(o.order_id) AS total_orders,
    SUM(p.price * o.quantity) AS total_sales
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
WHERE
    p.category = '전자기기'
GROUP BY
    p.category;

-- 생성된 뷰 조회
SELECT * FROM v_electronics_sales_status;
```

문제4: 기존 뷰에 정보 추가하여 수정하기

[문제]

문제 3에서 만들었던 `v_electronics_sales_status` 뷰가 매우 유용하다는 피드백을 받았다. 전략기획팀에서 총매출액뿐만 아니라, 평균 주문액(`average_order_value`)도 함께 보고 싶다는 추가 요청이 왔다.

기존 뷰를 삭제하지 말고, `ALTER VIEW`를 사용하여 `v_electronics_sales_status` 뷰의 정의를 수정해라. 총 주문 건수, 총매출액, 그리고 평균 주문액을 보여주도록 변경해야 한다.

[실행 결과]

category	total_orders	total_sales	average_order_value
전자기기	5	770000	154000.0000

[정답]

```
-- 뷰 수정
ALTER VIEW v_electronics_sales_status AS
SELECT
    p.category,
    COUNT(o.order_id) AS total_orders,
    SUM(p.price * o.quantity) AS total_sales,
    AVG(p.price * o.quantity) AS average_order_value
FROM
    orders o
JOIN
    products p ON o.product_id = p.product_id
WHERE
    p.category = '전자기기'
GROUP BY
    p.category;

-- 수정된 뷰 조회
SELECT * FROM v_electronics_sales_status;
```

정리

뷰(View) 소개

- 뷰(View)는 실제 데이터를 가지지 않는 '가상의 테이블'이며, 그 실체는 데이터베이스에 저장된 `SELECT` 쿼리문이다.
- 복잡하고 긴 쿼리를 매번 작성할 필요 없이, 뷰를 통해 간단하게 조회할 수 있어 편리하다.
- 사용자는 원본 테이블에 직접 접근할 필요 없이 뷰를 통해서만 제한된 데이터에 접근하게 되므로 보안성이 향상된다.
- 뷰를 조회할 때마다 저장된 `SELECT` 쿼리가 실행되므로, 데이터는 항상 원본 테이블의 최신 상태를 반영한다.
- 원본 테이블의 구조가 변경되어도 뷰의 정의만 수정하면 되므로, 논리적 데이터 독립성을 제공한다.

뷰 생성, 조회, 수정, 삭제

- `CREATE VIEW 뷰이름 AS SELECT 쿼리문;` 명령어를 사용하여 뷰를 생성한다.
- 생성된 뷰는 일반 테이블처럼 `SELECT` 문을 사용하여 조회할 수 있으며, `WHERE`나 `ORDER BY` 절도 사용 가능하다.
- `ALTER VIEW 뷰이름 AS SELECT 쿼리문;` 명령어를 사용하여 기존 뷰의 정의(저장된 쿼리)를 수정한다.
- `DROP VIEW 뷰이름;` 명령어를 사용하여 뷰를 삭제할 수 있으며, 뷰를 삭제해도 원본 테이블의 데이터는 전혀 손상되지 않는다.

뷰의 장점과 단점

- **장점**:
  - **편리성과 재사용성**: 복잡한 쿼리를 단순화하고 재사용하여 유지보수성을 높인다.
  - **보안성**: 특정 행이나 열만 노출하여 사용자별 데이터 접근 권한을 세밀하게 제어할 수 있다.
  - **논리적 데이터 독립성**: 테이블 구조가 변경되어도 뷰를 사용하는 응용 프로그램은 영향을 받지 않도록 보호하는 추상화 계층 역할을 한다.
- **단점 및 주의사항**:
  - **성능 문제**: 뷰의 쿼리가 단순해 보여도 실제로는 시스템에 큰 부하를 줄 수 있으며, 특히 뷰를 중첩하여 사용하면 성능 저하의 원인이 될 수 있다.
  - **업데이트 제약**: `JOIN`이나 집계 함수(`SUM`, `COUNT` 등)를 사용한 복잡한 뷰는 데이터를 수정(`INSERT`, `UPDATE`, `DELETE`)할 수 없다. 뷰는 기본적으로 '조회용'으로 사용하는 것이 원칙이다.