5. 데이터 변경 이력 설계2

#0.강의/2.데이터베이스로드맵/4.설계2

컬럼에 이전 값 보관 방식

지금까지는 현재 테이블에 "추적 정보"를 추가하는 방식이었다. 하지만 이 방식으로는 "이전 값"을 알 수 없었다.

먼저 이전 값을 보관하는 가장 단순한 방법부터 알아보자.

아이디어

아이디어는 간단하다. 현재 값과 이전 값을 각각 별도의 컬럼에 저장하는 것이다.

- `price`: 현재 가격
- `previous_price`: 이전 가격
- `price_changed_at`: 가격이 변경된 시점

테이블 설계

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
```

```
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    -- 현재 가격
    price INT NOT NULL,
    -- 이전 가격
    previous_price INT,
    price_changed_at DATETIME,

    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',

    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL
);
```

데이터 등록

상품을 처음 등록할 때는 아직 이전 가격이 없다.

```
INSERT INTO product (name, price, previous_price, price_changed_at,
stock_quantity, created_by, updated_by)
VALUES ('스마트폰 케이스', 15000, NULL, NULL, 100, 'admin_kim', 'admin_kim');

INSERT INTO product (name, price, previous_price, price_changed_at,
stock_quantity, created_by, updated_by)
VALUES ('무선 이어폰', 89000, NULL, NULL, 50, 'admin_lee', 'admin_lee');
```

```
SELECT product_id, name, price, previous_price, price_changed_at
FROM product;
```

[실행 결과]

product_id	name	price	previous_price	price_changed_at

1	스마트폰 케이스	15000	NULL	NULL
2	무선 이어폰	89000	NULL	NULL

가격 변경

가격을 변경할 때는 현재 가격을 이전 가격으로 옮기고, 새 가격을 입력한다.

```
UPDATE product
SET previous_price = price,
    price = 12000,
    price_changed_at = NOW(),
    updated_by = 'admin_park'
WHERE product_id = 1;
```

```
SELECT product_id, name, price, previous_price, price_changed_at
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	previous_price	price_changed_at
1	스마트폰 케이스	12000	15000	2026-03-01 10:00:00

이제 이전 가격을 알 수 있다. 현재 가격은 12,000원이고, 이전 가격은 15,000원이었다.

추가로 `price_changed_at` 컬럼을 통해서 가격이 변경된 시점도 알 수 있다.

한 번 더 가격 변경

가격을 한 번 더 변경해보자.

```sql
UPDATE product
SET previous_price = price,
    price = 10000,
    price_changed_at = NOW(),
    updated_by = 'admin_kim'
WHERE product_id = 1;
```

```sql
SELECT product_id, name, price, previous_price, price_changed_at
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	previous_price	price_changed_at
1	스마트폰 케이스	10000	12000	2026-03-15 14:00:00

문제가 발생했다. 현재 가격 10,000원과 직전 가격 12,000원은 알 수 있지만, 최초 가격 15,000원은 사라졌다.

단점

이 방식은 "바로 직전 값" 하나만 보관할 수 있다. 두 번 이상 변경되면 과거 값이 사라진다.

```sql
-- 가격 변경 이력을 모두 보여주세요
SELECT product_id, name,
       price AS current_price,
       previous_price AS one_before,
       '(최초 가격, 15000): 사라짐' AS two_before
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	current_price	one_before	two_before
1	스마트폰 케이스	10000	12000	(최초 가격, 15000): 사라짐

여러 컬럼을 추적해야 한다면?

가격뿐만 아니라 재고, 상태 등도 이전 값을 추적해야 한다면 어떻게 될까?

```
-- 이런 식으로 컬럼이 폭발적으로 늘어난다
CREATE TABLE product_example (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(200),

    -- 가격 추적
    price INT,
    previous_price INT,
    price_changed_at DATETIME,

    -- 재고 추적
    stock_quantity INT,
    previous_stock_quantity INT,
    stock_changed_at DATETIME,

    -- 상태 추적
    status VARCHAR(20),
    previous_status VARCHAR(20),
    status_changed_at DATETIME

    -- 추적할 컬럼이 늘어날 때마다 3개씩 컬럼이 추가된다...
);
```

컬럼이 너무 많아진다. 관리가 어려워지고, 테이블 구조가 복잡해진다.

컬럼에 이전 값 보관 방식 - 정리

장점

- 구현이 매우 간단하다.
- 조회가 빠르다 (조인 없이 한 테이블에서 조회).
- 직전 값만 필요한 경우 유용하다.

단점

- 직전 값 하나만 보관할 수 있다.
- 두 번 이상 변경되면 과거 이력이 사라진다.
- 추적할 컬럼이 많아지면 테이블 구조가 복잡해진다.
- 특정 시점의 값을 조회할 수 없다.

언제 사용하면 좋을까?

- 변경이 거의 없는 데이터 (예: 회원 이름)
- 직전 값만 알면 되는 경우
- 빠른 조회가 중요하고, 전체 이력이 필요 없는 경우

실무에서 이전 값 보관만으로 충분한 경우는 드물다. 대부분은 전체 변경 이력이 필요하다. 다음 수업에서는 행 자체를 보관하는 방법을 알아보겠다.

> ✨ **컬럼에 이전 값 보관 - SCD Type 3**
> SCD는 Slowly Changing Dimension의 약자로, 데이터 웨어하우스 분야에서 사용하는 용어다. 천천히 변경되는 데이터를 어떻게 관리할 것인가에 대한 여러 가지 방법론이 있다. 그중 Type 3는 "이전 값을 별도 컬럼에 저장하는 방식"이다. 이번에 사용한 방식이 SCD Type 3 방식이다.

현재 테이블로 이력 관리 - 시작

이전 컬럼 하나만 저장하는 방식은 한계가 명확했다. 전체 변경 이력을 관리하려면 결국 "행(Row) 자체"를 보관해야 한다. 가장 직관적인 방법은 현재 테이블에 모든 이력을 함께 저장하는 것이다.

아이디어

데이터를 수정할 때 기존 행을 UPDATE하지 않고, 새로운 행을 INSERT한다. 이렇게 하면 모든 변경 이력이 행으로 남는다.

그리고 `is_current` 필드를 사용해서 최신 데이터를 구분한다.

테이블 설계

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    history_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    is_current BOOLEAN NOT NULL DEFAULT TRUE,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,

    INDEX idx_product_id (product_id),
    INDEX idx_is_current (is_current),
    INDEX idx_product_id2 (product_id, created_at)
);
```

주요 변경 사항은 다음과 같다.

- `history_id`: 각 이력 행의 고유 ID (Primary Key)
- `product_id`: 실제 상품 ID (같은 상품의 이력들은 같은 product_id를 가진다)
- `is_current`: 현재 유효한 데이터인지 여부 (TRUE면 최신 데이터)

여기서는 PK가 `history_id`이기 때문에 같은 `product_id`도 등록할 수 있다.

데이터 등록

상품을 처음 등록한다.

```
INSERT INTO product (product_id, name, price, stock_quantity, status,
is_current, created_by, created_at)
VALUES (1, '스마트폰 케이스', 15000, 100, 'ACTIVE', TRUE, 'admin_kim', '2026-01-15
10:00:00');
```

```
INSERT INTO product (product_id, name, price, stock_quantity, status,
is_current, created_by, created_at)
VALUES (2, '무선 이어폰', 89000, 50, 'ACTIVE', TRUE, 'admin_lee', '2026-01-15
10:05:00');
```

- `created_at`: 이후에 사용할 예제를 위해서 시간 정보를 직접 입력했다.

```
SELECT history_id, product_id, name, price, is_current, created_at
FROM product;
```

[실행 결과]

history_id	product_id	name	price	is_current	created_at
1	1	스마트폰 케이스	15000	1	2026-01-15 10:00:00
2	2	무선 이어폰	89000	1	2026-01-15 10:05:00

- `history_id`가 PK이다. `history_id`는 자동증가하는 값이다.
- `product_id`는 상품 ID를 직접 입력했다.

데이터 변경

스마트폰 케이스의 가격을 15000 ➜ 12000으로 변경해보자.

이 방식에서 가격을 변경할 때는 두 가지 작업이 필요하다.

1. 기존 행의 `is_current`를 FALSE로 변경
2. 새로운 행을 INSERT

```
-- 1. 기존 행을 과거 데이터로 변경
UPDATE product
SET is_current = FALSE
WHERE product_id = 1 AND is_current = TRUE;
```

```
-- 2. 새로운 행 추가
INSERT INTO product (product_id, name, price, stock_quantity, status,
is_current, created_by, created_at) VALUES (1, '스마트폰 케이스', 12000, 100,
'ACTIVE', TRUE, 'admin_park', '2026-03-01 10:00:00');
```

```
SELECT history_id, product_id, name, price, is_current, created_at, created_by
FROM product
WHERE product_id = 1
ORDER BY history_id;
```

[실행 결과]

history_id	product_id	name	price	is_current	created_at	create
1	1	스마트폰 케이스	15000	0	2026-01-15 10:00:00	admin
3	1	스마트폰 케이스	12000	1	2026-03-01 10:00:00	admin

두 개의 행이 있다. `is_current = 0`인 행은 과거 데이터이고, `is_current = 1`인 행이 현재 데이터다.

현재 데이터만 조회

`is_current=TRUE`를 통해 현재 유효한 데이터만 조회할 수 있다. 스마트폰의 현재 유효한 데이터를 확인해보자.

```
SELECT history_id, product_id, name, price, stock_quantity, status, is_current
FROM product
WHERE is_current = TRUE
  AND product_id = 1;
```

[실행 결과]

history_id	product_id	name	price	stock_quantity	status	is_cu

3	1	스마트폰 케이스	12000	100	ACTIVE	1

한 번 더 변경

이번에는 스마트폰의 가격을 12000 ➜ 10000으로 변경해보자. 그리고 재고도 100 ➜ 95로 변경하자.

```
-- 1. 기존 행을 과거 데이터로 변경
UPDATE product
SET is_current = FALSE
WHERE product_id = 1 AND is_current = TRUE;

-- 2. 새로운 행 추가
INSERT INTO product (product_id, name, price, stock_quantity, status,
is_current, created_by, created_at) VALUES (1, '스마트폰 케이스', 10000, 95,
'ACTIVE', TRUE, 'admin_kim', '2026-03-15 14:00:00');
```

```
SELECT history_id, product_id, name, price, stock_quantity, is_current,
created_at
FROM product
WHERE product_id = 1
ORDER BY history_id;
```

[실행 결과]

history_id	product_id	name	price	stock_quantity	is_current	created
1	1	스마트폰 케이스	15000	100	0	2026-01 10:00:00
3	1	스마트폰 케이스	12000	100	0	2026-03 10:00:00
4	1	스마트폰 케이스	10000	95	1	2026-03 14:00:00

이제 모든 변경 이력이 남는다. 가격이 15,000 → 12,000 → 10,000으로 변경된 것을 모두 확인할 수 있다.
재고도 마지막에 100 ➜ 95로 변경된 것을 확인할 수 있다.

과거 데이터만 조회

`is_current=FALSE` 컬럼을 통해 과거 데이터만 조회할 수 있다.

```
SELECT history_id, product_id, name, price, stock_quantity, status, is_current
FROM product
WHERE is_current = FALSE;
```

[실행 결과]

history_id	product_id	name	price	stock_quantity	status	is_cu
1	1	스마트폰 케이스	15000	100	ACTIVE	0
3	1	스마트폰 케이스	12000	100	ACTIVE	0

특정 상품의 전체 이력 조회

```
SELECT history_id, product_id, name, price, stock_quantity, created_at,
created_by,
       CASE WHEN is_current THEN '현재' ELSE '과거' END AS data_status
FROM product
WHERE product_id = 1
ORDER BY created_at;
```

[실행 결과]

history_id	product_id	name	price	stock_quantity	created_at	created_by

1	1	스마트폰 케이스	15000	100	2026-01-15 10:00:00	admin_kim
3	1	스마트폰 케이스	12000	100	2026-03-01 10:00:00	admin_park
4	1	스마트폰 케이스	10000	95	2026-03-15 14:00:00	admin_kim

장점

이 방식의 장점은 명확하다.

1. **전체 이력 보관**: 모든 변경 이력이 행으로 남는다.
2. **이전 값 확인 가능**: 언제든 과거 값을 조회할 수 있다.
3. **구현이 단순하다**: 별도의 이력 테이블 없이 하나의 테이블로 관리한다.

현재 테이블로 이력 관리 - 단점 1

단점 - 시점 조회의 어려움

하지만 이 방식은 특정 시점의 데이터를 조회하기 어렵다는 문제가 있다.

먼저 단일 상품을 기준으로 "2026년 3월 10일 기준으로 가격이 얼마였나요?"라는 질문에 답해보자.
여기서는 스마트폰 케이스(`product_id=1`)을 조회해보자.

2026년 3월 10일 시점의 데이터를 찾아야 한다. 먼저 해당 상품의 전체 이력을 확인해보자.

```
SELECT history_id, product_id, name, price, created_at, is_current
FROM product
WHERE product_id = 1
ORDER BY created_at DESC
```

history_id	product_id	name	price	created_at	is_current
4	1	스마트폰 케이스	10000	2026-03-15 14:00:00	1
3	1	스마트폰 케이스	12000	2026-03-01 10:00:00	0
1	1	스마트폰 케이스	15000	2026-01-15 10:00:00	0

`created_at` 값을 기준으로 각 데이터가 사용된 기간을 정리하면 다음과 같다. (시간은 제외했다)

- `history_id=4` : 2026-03-15 ➜ `[2026-03-15 ~ 현재]`
- `history_id=3` : 2026-03-01 ➜ `[2026-03-01 ~ 2026-03-15 이전]`
- `history_id=1` : 2026-01-15 ➜ `[2026-01-15 ~ 2026-03-01 이전]`

따라서

- `history_id=4`의 경우 `[2026-03-15 ~ 현재]` 데이터 이므로 2026년 3월 10일 기준을 만족하지 않는다.
- `history_id=3`의 경우 `[2026-03-01 ~ 2026-03-15 이전]` 데이터이므로 2026년 3월 10일 기준에 만족한다.
- `history_id=1`의 경우 `[2026-01-15 ~ 2026-03-01 이전]` 데이터이므로 2026년 3월 10일 기준을 만족하지 않는다.

정리하면 `history_id=3`이 2026년 3월 10일에 사용된 데이터이므로 이 데이터를 선택하면 된다.
쿼리로는 간단하게 다음과 같이 나타낼 수 있다.

```sql
-- 2026-03-10 기준 가격 조회
SELECT history_id, product_id, name, price, created_at, is_current
FROM product
WHERE product_id = 1
  AND created_at <= '2026-03-10 23:59:59'
ORDER BY created_at DESC
```

- 날짜 내림차순으로 정렬한 다음 생성일이 `2026-03-10`일 보다 같거나 작은 날짜를 찾아보자.

> 참고: 예시를 단순화 하기 위해 예시들에 밀리초는 없다고 가정하겠다.

[실행 결과]

history_id	product_id	name	price	created_at	is_current
3	1	스마트폰 케이스	12000	2026-03-01 10:00:00	0
1	1	스마트폰 케이스	15000	2026-01-15 10:00:00	0

> **참고**
> 조회 결과 `history_id=3`의 현재 `is_current`는 0(False)으로 나온다.
> 2026년 3월 10일 당시에는 이 값이 1(True)이었을 것이다.

- 생성일이 2026-03-10보다 같거나 작은 날짜를 찾으니 2가지 데이터가 나온다. 내림차순에서 가장 상단에 있는 2026-03-01 데이터가 우리가 찾는 데이터이다. 이 데이터는 2026-03-01 ~ 2026-03-15 이전까지 사용된 데이터이다. 그 이전 데이터들은 모두 제거해야 한다.

참고

- `history_id=4`의 경우 `[2026-03-15 ~ 현재]` 데이터 이므로 2026년 3월 10일 기준을 만족하지 않는다.
- `history_id=3`의 경우 `[2026-03-01 ~ 2026-03-15 이전]` 데이터이므로 2026년 3월 10일 기준에 만족한다.
- `history_id=1`의 경우 `[2026-01-15 ~ 2026-03-01 이전]` 데이터이므로 2026년 3월 10일 기준을 만족하지 않는다.

그 이전에 데이터들은 제거하고, 딱 하나의 데이터만 찾으려면 날짜 내림차순이기 때문에 `LIMIT 1`을 사용하면 된다.

```sql
-- 2026-03-10 기준 가격 조회
SELECT history_id, product_id, name, price, created_at, is_current
FROM product
WHERE product_id = 1
  AND created_at <= '2026-03-10 23:59:59'
ORDER BY created_at DESC
LIMIT 1;
```

- 따라서 날짜 내림차순으로 정렬한 다음 `2026-03-10`일 보다 같거나 작은 날짜중 처음으로 등장한 날짜를 하나만 찾으면 된다.

[실행 결과]

history_id	product_id	name	price	created_at	is_current
3	1	스마트폰 케이스	12000	2026-03-01 10:00:00	0

3월 10일 기준 가격은 12,000원이었다. 단일 상품의 경우 특정 시점의 조회는 간단하다.

현재 테이블로 이력 관리 - 단점 2

전체 통계에서의 성능 문제

앞서 본 것 처럼 단일 상품의 시점 조회는 괜찮다. 하지만 전체 상품의 특정 시점 통계를 내야 한다면 어떻게 될까?

"2026년 3월 10일 기준, 모든 상품의 총 재고 수량은?"

우선 모든 데이터를 확인해보자.

```
SELECT history_id, product_id, name, created_at, stock_quantity
FROM product
ORDER BY created_at DESC;
```

history_id	product_id	name	created_at	stock_quantity
4	1	스마트폰 케이스	2026-03-15 14:00:00	95
3	1	스마트폰 케이스	2026-03-01 10:00:00	100
2	2	무선 이어폰	2026-01-15 10:05:00	50
1	1	스마트폰 케이스	2026-01-15 10:00:00	100

2026년 3월 10일 기준 상품은 다음과 같다.

- **스마트폰 케이스**: `history_id=3`, 재고:100
- **무선 이어폰**: `history_id=2`, 재고: 50

각 상품별 2026년 3월 10일 기준 재고를 모두 구해서 합하는 쿼리는 다음과 같다.

```
-- 각 상품별로 해당 시점의 최신 데이터를 찾아야 한다
SELECT SUM(p.stock_quantity) AS total_stock
FROM product p
INNER JOIN (
    SELECT product_id, MAX(created_at) AS max_created_at
    FROM product
    WHERE created_at <= '2026-03-10 23:59:59'
    GROUP BY product_id
) latest ON p.product_id = latest.product_id
        AND p.created_at = latest.max_created_at;
```

[실행 결과]

total_stock
150

이 쿼리는 서브쿼리(Subquery)와 조인(Join), 그리고 집계 함수(Aggregate Function)가 섞여 있어 처음 보면 복잡해 보일 수 있다. 우리가 원하는 것은 **"2026년 3월 10일 시점에 각 상품의 가장 마지막 상태"**를 찾아내는 것이다. 이 쿼리가 만들어진 과정을 단계별로 나누어 살펴보자.

1단계: 특정 시점 이전의 데이터만 필터링하기

가장 먼저 해야 할 일은 타임머신을 타고 과거로 돌아가는 것이다. 기준 날짜인 '2026년 3월 10일' 이후에 생성된 데이터는 미래의 일이므로 모두 무시해야 한다. 상품을 찾는 시점인 `2026-03-10`이나 그 이전 시점만 조회해보자.

```
SELECT history_id, product_id, name, created_at, stock_quantity
FROM product
WHERE created_at <= '2026-03-10 23:59:59'
ORDER BY created_at DESC;
```

[실행 결과]

history_id	product_id	name	created_at	stock_quantity
3	1	스마트폰 케이스	2026-03-01 10:00:00	100
2	2	무선 이어폰	2026-01-15 10:05:00	50
1	1	스마트폰 케이스	2026-01-15 10:00:00	100

결과를 보면 3월 15일에 생성된 데이터(`history_id=4`)는 제외되었다.

하지만 `product_id = 1`인 상품의 데이터가 2개(1월 15일, 3월 1일) 존재한다. 왜냐하면 `WHERE created_at <= '2026-03-10 23:59:59'` 이 검색조건은 `2026-03-10`을 포함한 이전의 데이터를 모두 조회하기 때문이다.

이전에 하나의 상품을 조회할 때는 단순하게 `LIMIT 1`을 적용해서 가장 상단에 있는 데이터를 조회하면 되었다. 하지만 여러 상품이 섞여 있는 경우에는 이 방법을 사용할 수 없다. (여기서는 무선 이어폰의 재고도 찾아야 한다.)

2단계: 상품별로 가장 마지막 시간 구하기

이제 필터링된 데이터 중에서, 각 상품별로 가장 최근의 시간(`MAX(created_at)`)을 찾아야 한다. 이것이 바로 그 시점의 '유효한 데이터'이기 때문이다.

스마트폰 케이스의 경우 다음 두 시간중에 가장 최근의 시간인 2026-03-01을 찾으면 된다.

- 2026-03-01 10:00:00
- 2026-01-15 10:00:00

`GROUP BY`를 사용해서 각 상품별로 묶고 가장 큰 날짜를 구한다.

```sql
SELECT product_id, MAX(created_at) AS max_created_at
FROM product
WHERE created_at <= '2026-03-10 23:59:59'
GROUP BY product_id;
```

[실행 결과]

product_id	max_created_at
1	2026-03-01 10:00:00

2	2026-01-15 10:05:00

이제 우리는 각 상품별로 3월 10일 기준, 어떤 시간의 데이터가 '진짜'인지 알게 되었다.

- 상품 1번(스마트폰 케이스)은 3월 1일 10시 데이터가 최신이다.
- 상품 2번(무선 이어폰)은 1월 15일 10시 5분 데이터가 최신이다.

3단계: 원본 테이블과 조인하여 재고 정보 가져오기

2단계에서 구한 것은 단순히 `product_id`와 `시간` 정보뿐이다. 실제 `stock_quantity(재고)` 정보를 알기 위해서는 이 정보를 바탕으로 다시 원본 `product` 테이블과 조인(Join)해야 한다.

이때 **'상품 ID가 같고' AND '생성 시간이 같은'** 행을 찾아야 정확한 이력 데이터를 가져올 수 있다.

```
SELECT p.product_id, p.name, p.stock_quantity, p.created_at
FROM product p
INNER JOIN (
    SELECT product_id, MAX(created_at) AS max_created_at
    FROM product
    WHERE created_at <= '2026-03-10 23:59:59'
    GROUP BY product_id
) latest ON p.product_id = latest.product_id
        AND p.created_at = latest.max_created_at;
```

[실행 결과]

product_id	name	stock_quantity	created_at
1	스마트폰 케이스	100	2026-03-01 10:00:00
2	무선 이어폰	50	2026-01-15 10:05:00

드디어 3월 10일 기준의 각 상품별 유효한 행을 하나씩 뽑아냈다.

4단계: 최종 집계

이제 각 상품의 재고 수량(`stock_quantity`)을 모두 더하기만 하면 된다.

```
SELECT SUM(p.stock_quantity) AS total_stock
FROM product p
INNER JOIN (
    SELECT product_id, MAX(created_at) AS max_created_at
    FROM product
    WHERE created_at <= '2026-03-10 23:59:59'
    GROUP BY product_id
) latest ON p.product_id = latest.product_id
        AND p.created_at = latest.max_created_at;
```

[실행 결과]

total_stock
150

상품 1의 재고(100개)와 상품 2의 재고(50개)가 합쳐져서 최종 결과 150개가 나왔다.

이처럼 **'현재 테이블에 이력을 함께 쌓는 방식'**은 데이터 입력은 쉽지만, 특정 시점의 데이터를 조회하거나 통계를 낼 때 쿼리가 매우 복잡해지고 성능(Performance) 이슈가 발생할 수 있다. 특히 각 상품별 특정 시점을 찾기 위해 특정 시점 이전의 과거 전체 이력을 찾아야 하는 서브쿼리가 필요하고, 또 추가적인 조인이 필요하다. 이런 쿼리는 성능을 최적화하기 어렵다.

만약 상품이 10만 개이고, 각 상품이 평균 10번씩 변경되었다면 테이블에는 100만 개의 행이 있다. 이 중에서 각 상품의 특정 시점 데이터를 찾으려면 서브쿼리에서 전체 테이블을 스캔해야 한다. 그리고 테이블의 크기가 커질수록 이 `GROUP BY`와 `JOIN` 연산은 데이터베이스에 큰 부하를 주게 된다.

결과적으로 이 방식에는 3가지 문제가 있다.

1. 시점 조회와 통계 쿼리 작성의 복잡함
2. 시점 조회와 통계 쿼리의 성능 문제
3. 한 테이블에 너무 많은 데이터 보유

정리

현재 테이블에 이력을 함께 저장하는 방식은 전체 이력을 보관할 수 있지만, 시점 조회와 통계 쿼리에서 성능 문제가 발생한다. 다음 수업에서는 이 문제를 해결하는 방법을 알아보겠다.

현재 테이블로 이력 관리 - 유효 기간

이전 수업에서 `is_current` 플래그만으로는 시점 조회가 어렵다는 것을 확인했다. 이 문제를 해결하기 위해 "유효 기간"을 추가하는 방법이 있다.

유효 기간 아이디어

각 행에 "이 데이터가 유효한 기간"을 명시한다.

- `valid_from`: 이 데이터가 유효해진 시점
- `valid_to`: 이 데이터가 더 이상 유효하지 않게 된 시점 (현재 데이터는 NULL 또는 먼 미래 날짜)

테이블 설계

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    history_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    -- 유효 기간
    valid_from DATETIME NOT NULL,
    valid_to DATETIME NOT NULL DEFAULT '9999-12-31 23:59:59',
    is_current BOOLEAN NOT NULL DEFAULT TRUE,
    created_by VARCHAR(100) NOT NULL,

    INDEX idx_product_id_valid_range (product_id, valid_from, valid_to),
    INDEX idx_valid_range (valid_from, valid_to),
    INDEX idx_is_current (is_current)
);
```

- `valid_to`의 기본값을 '9999-12-31 23:59:59'로 설정했다. 이렇게 하면 현재 유효한 데이터를 쉽게 식별할

수 있다.

- `idx_product_id_valid_range`, `idx_valid_range` 인덱스를 추가해서 유효 기간 검색을 빠르게 한다.

데이터 등록

```
INSERT INTO product (product_id, name, price, stock_quantity, status,
valid_from, valid_to, is_current, created_by)
VALUES (1, '스마트폰 케이스', 15000, 100, 'ACTIVE', '2026-01-10 10:00:00',
'9999-12-31 23:59:59', TRUE, 'admin_kim');

INSERT INTO product (product_id, name, price, stock_quantity, status,
valid_from, valid_to, is_current, created_by)
VALUES (2, '무선 이어폰', 89000, 50, 'ACTIVE', '2026-01-10 10:05:00', '9999-12-31
23:59:59', TRUE, 'admin_lee');
```

```
SELECT history_id, product_id, name, price, valid_from, valid_to, is_current
FROM product;
```

[실행 결과]

history_id	product_id	name	price	valid_from	valid_to	is_cu
1	1	스마트폰 케이스	15000	2026-01-10 10:00:00	9999-12-31 23:59:59	1
2	2	무선 이어폰	89000	2026-01-10 10:05:00	9999-12-31 23:59:59	1

데이터 변경

2026-01-12일(현재로 가정) 스마트폰 케이스의 가격을 15000 ➜ 12000으로 변경해보자.

이 방식에서 가격을 변경할 때는 세 가지 작업이 필요하다.

1. 기존 행의 `valid_to`를 현재 시점으로 변경

2. 기존 행의 `is_current`를 `FALSE`로 변경
3. 새로운 행을 INSERT

```
-- 변경 시점 (현재로 가정)
SET @change_time = '2026-01-12 10:00:00';

-- 1. 기존 행의 유효 기간 종료
UPDATE product
SET valid_to = @change_time,
    is_current = FALSE
WHERE product_id = 1 AND is_current = TRUE;

-- 2. 새로운 행 추가
INSERT INTO product (product_id, name, price, stock_quantity, status,
valid_from, valid_to, is_current, created_by)
VALUES (1, '스마트폰 케이스', 12000, 100, 'ACTIVE', @change_time, '9999-12-31
23:59:59', TRUE, 'admin_park');
```

스마트폰 조회

```
SELECT history_id, product_id, name, price, valid_from, valid_to, is_current
FROM product
WHERE product_id = 1
ORDER BY valid_from;
```

[실행 결과]

history_id	product_id	name	price	valid_from	valid_to	is_cu
1	1	스마트폰 케이스	15000	2026-01-10 10:00:00	2026-01-12 10:00:00	0
3	1	스마트폰 케이스	12000	2026-01-12 10:00:00	9999-12-31 23:59:59	1

`valid_from`, `valid_to` 컬럼 덕분에 이제 각 행이 언제부터 언제까지 유효한지 명확하다.

- 첫 번째 행: 2026-01-10 ~ 2026-01-12 동안 유효 (가격 15,000원)

- 두 번째 행: 2026-01-12 ~ 현재까지 유효 (가격 12,000원)

참고: 9999-12-31은 항상 지금 시점을 포함한다.

한 번 더 변경

2026-01-14일 (현재로 가정) 스마트폰 케이스의 가격을 12000 ➜ 10000으로 변경하고, 재고를 100 ➜ 95로 변경한다.

```
SET @change_time = '2026-01-14 14:00:00';

UPDATE product
SET valid_to = @change_time,
    is_current = FALSE
WHERE product_id = 1 AND is_current = TRUE;

INSERT INTO product (product_id, name, price, stock_quantity, status,
valid_from, valid_to, is_current, created_by)
VALUES (1, '스마트폰 케이스', 10000, 95, 'ACTIVE', @change_time, '9999-12-31
23:59:59', TRUE, 'admin_kim');
```

```
SELECT history_id, product_id, name, price, stock_quantity, valid_from,
valid_to, is_current
FROM product
WHERE product_id = 1
ORDER BY valid_from;
```

[실행 결과]

history_id	product_id	name	price	stock_quantity	valid_from	valid_to	is_
1	1	스마트폰 케이스	15000	100	2026-01-10 10:00:00	2026-01-12 10:00:00	0
3	1	스마트폰 케이스	12000	100	2026-01-12 10:00:00	2026-01-14 14:00:00	0

4	1	스마트폰 케이스	10000	95	2026-01-14 14:00:00	9999-12-31 23:59:59	1

특정 시점 조회 - 훨씬 간단해진 쿼리

이제 특정 시점을 조회해보자.

"2026년 1월 13일 기준 스마트폰 케이스의 가격"을 조회해보자.

```
SELECT product_id, name, price, valid_from, valid_to
FROM product
WHERE product_id = 1
  AND '2026-01-13 23:59:59' >= valid_from
  AND '2026-01-13 23:59:59' < valid_to;
```

[실행 결과]

product_id	name	price	valid_from	valid_to
1	스마트폰 케이스	12000	2026-01-12 10:00:00	2026-03-14 14:00:00

서브쿼리 없이 단순한 범위 조건만으로 특정 시점의 데이터를 조회할 수 있다. 인덱스도 효율적으로 사용된다.

현재 시점 조회

`valid_to`의 기본 값을 `9999-12-31 23:59:59`로 지정해둔 덕분에 현재 시점에 유효한 상품도 쉽게 검색할 수 있다.

```
SELECT product_id, name, price, valid_from, valid_to
FROM product
WHERE now() >= valid_from
  AND now() < valid_to
ORDER BY product_id;
```

[실행 결과]

product_id	name	price	valid_from	valid_to
1	스마트폰 케이스	10000	2026-01-14 14:00:00	9999-12-31 23:59:59
2	무선 이어폰	89000	2026-01-10 10:05:00	9999-12-31 23:59:59

전체 통계 - 성능 개선

"2026년 1월 13일 기준, 모든 상품의 총 재고 수량"을 조회해보자.

```
SELECT SUM(stock_quantity) AS total_stock
FROM product
WHERE '2026-01-13 23:59:59' >= valid_from
  AND '2026-01-13 23:59:59' < valid_to;
```

[실행 결과]

total_stock
150

과거와는 비교할 수 없을 정도로 쿼리가 매우 단순해졌다. 서브쿼리나 조인 없이 단순 WHERE 조건만으로 해결된다.
또한 범위 컬럼에 인덱스를 사용해서 성능도 과거와 비교할 수 없을 정도로 빠르다.

valid_to, is_current 컬럼 중복 문제

테이블 설계를 유심히 본 분이라면 이런 의문이 들 수 있다.

"`valid_to`가 `9999-12-31`인 데이터를 찾으면 그게 현재 유효한 데이터인데, 굳이 `is_current` 컬럼을 따로 만들어서 저장 공간을 낭비해야 하나?"

결론부터 말하자면 **데이터의 중복이 맞다.**

정규화 이론에 따르면 중복 데이터는 제거해야 한다. 하지만 실무에서는 **개발의 편의성**과 **성능 최적화**를 위해 의도적으로 중복 데이터를 허용하기도 하는데, 이를 **반정규화(Denormalization)**라고 한다.

`is_current` 컬럼을 추가함으로써 얻을 수 있는 실무적 이점은 다음과 같다.

1. 쿼리의 직관성 (가독성)
가장 큰 이유는 개발자가 쿼리를 작성할 때 직관적이기 때문이다. 현재 유효한 데이터를 조회하는 두 쿼리를 비교해보자.

```
-- 1. valid_to를 사용하는 경우 (매직 넘버 사용)
SELECT *
FROM product
WHERE valid_to = '9999-12-31 23:59:59';

-- 2. is_current를 사용하는 경우
SELECT *
FROM product
WHERE is_current = TRUE;
```

`valid_to`를 사용할 때는 `9999-12-31`이라는 특정한 날짜(매직 넘버)를 정확히 입력해야 한다. 만약 실수로 날짜를 다르게 입력하거나, 팀 내부 정책이 바뀌어 종료 날짜 기준이 변경된다면 관련된 모든 쿼리를 수정해야 한다.
반면 `is_current = TRUE`는 "현재 유효한 데이터"라는 비즈니스 의미가 명확하며, 날짜 정책이 바뀌어도 쿼리를 수정할 필요가 없다.

2. 성능상 이점 (인덱스 효율)
데이터베이스 인덱스 관점에서 보면 **동등 비교(`=`)**가 **범위 검색(`>`)**보다 효율적일 때가 많다.

대부분의 조회 쿼리는 현재 유효한 상품을 찾는 경우다.
만약 `product_id` 별로 현재 데이터를 빨리 찾고 싶다면 복합 인덱스를 구성할 수 있다.

- **구성 A (`valid_to` 활용)**: `(product_id, valid_to)`
- **구성 B (`is_current` 활용)**: `(product_id, is_current)`

구성 A의 경우 `valid_to`가 범위 값이나 특정 날짜 값으로 인덱스 크기가 상대적으로 크다.
반면 구성 B의 `is_current`는 `TRUE/FALSE` 1비트 정보만 가지므로 인덱스 크기가 작고, 옵티마이저가 판단하기에도 훨씬 단순하다. 특히 수억 건의 데이터 중 "현재 데이터"만 빠르게 필터링해야 할 때 `is_current` 컬럼에 인덱스

를 걸면 매우 빠른 조회가 가능하다.

3. 미래의 확장성

지금은 `valid_to = 9999...`가 현재 데이터를 의미하지만, 미래에 비즈니스 요건이 바뀔 수 있다.
예를 들어 "삭제된 데이터"는 `is_deleted`로 관리하거나, "임시 저장" 상태가 추가될 수도 있다. 날짜만으로는 "논리적으로 현재 유효한 상태(Active)"인지 판단하기 복잡해질 수 있다. 이때 `is_current` 플래그는 날짜와 무관하게 **"서비스에 노출 중인 최신 데이터"**라는 의미를 명확하게 유지해준다.

정리하자면, `is_current`는 약간의 저장 공간을 사용하여 **쿼리의 단순함, 안전성, 성능**을 사는 가성비 좋은 설계다.

장점 정리

유효 기간 필드의 장점은 다음과 같다.

1. **시점 조회가 쉽다**: 범위 조건만으로 특정 시점의 데이터를 조회할 수 있다.
2. **통계 쿼리 성능이 좋다**: 서브쿼리 없이 단순 조건으로 조회할 수 있다.
3. **전체 이력 보관**: 모든 변경 이력이 유효 기간과 함께 보관된다.
4. **인덱스 활용이 좋다**: 범위 검색에 인덱스를 효과적으로 사용할 수 있다.

단점

하지만 이 방식에도 단점이 있다.

단점 1: 변경 시 UPDATE가 필요하다

데이터를 변경할 때마다 이전 행의 `valid_to`를 UPDATE해야 한다. 이력 테이블이 완전히 불변(Immutable)하지 않다.

```
-- 변경할 때마다 이 UPDATE가 필요하다
UPDATE product
SET valid_to = @change_time,
    is_current = FALSE
WHERE product_id = 1 AND is_current = TRUE;
```

단점 2: 트랜잭션 관리가 필요하다

UPDATE와 INSERT가 원자적으로 실행되어야 한다. 중간에 실패하면 데이터 정합성이 깨질 수 있다.

```
START TRANSACTION;
-- UPDATE 이전 행
-- INSERT 새 행
COMMIT;
```

단점 3: 데이터가 계속 쌓인다

현재 테이블에 이력까지 모두 쌓이므로 테이블 크기가 계속 커진다.

```
-- 현재 데이터만 필요한 일반적인 조회에서도
-- 많은 이력 데이터를 함께 스캔해야 한다
SELECT * FROM product WHERE is_current = TRUE;
```

앞서 설명했듯이 만약 상품이 10만 개이고, 각 상품이 평균 10번씩 변경되었다면 테이블에는 100만 개의 행이 만들어진다.
실무에서 가장 많이 사용하는 조회는 "현재 데이터"다. 보통 100번 조회 중 99번은 현재 데이터를 조회하고, 1번 정도만 이력을 조회한다. 그런데 현재 테이블에 이력이 함께 있으면 현재 데이터 조회에도 영향을 준다. 현재 데이터와 이력 데이터를 분리해서 현재 데이터를 가볍게 유지하는 방법이 필요하다.

이제 남은 1가지 문제를 해결해보자.

1. 시점 조회와 통계 쿼리 작성의 복잡함 ➜ 해결
2. 시점 조회와 통계 쿼리의 성능 문제 ➜ 해결
3. 한 테이블에 너무 많은 데이터 보유 ➜ 미해결

지금까지 이력 데이터를 한 테이블에 관리하는 방법을 알아보았다. 하지만 앞서 언급했듯 데이터가 계속 쌓여 테이블이 비대해지는 문제는 여전히 남아있다. 다음 강의에서는 이력 테이블을 별도로 분리하여 이 문제를 해결해보자.

> ✨ **유효기간 보관 - SCD Type 2**
>
> **SCD Type 2**는 데이터 변경 시 **기존 행을 업데이트로 덮어쓰지 않고, 새 행을 추가(버전 생성)** 하여 이력을 보관하는 방식이다. 각 버전 행에는 보통 `valid_from`, `valid_to` 같은 **유효 기간 컬럼**을 둬서 어떤 시점에 어떤 값이 유효했는지 조회할 수 있게 한다.

- `valid_from`: 이 데이터가 유효해진 시점
- `valid_to`: 이 데이터가 더 이상 유효하지 않게 된 시점 (현재 데이터는 NULL 또는 먼 미래 날짜)

전체 행 스냅샷 이력 테이블 - 시작

지금까지 배운 방식은 현재 테이블에 이력을 함께 저장하는 방식이었다. 실무에서 가장 많이 사용하는 패턴은 **현재 테이블과 이력 테이블을 분리**하는 것이다.

분리가 필요한 이유

실무에서 데이터 조회 패턴을 분석해보면 보통은 다음과 같다. (물론 비즈니스 상황에 따라 다르다.)

- **현재 데이터 조회**: 99%
- **이력 조회**: 1%

대부분의 조회는 현재 데이터를 조회한다. 상품 목록, 주문 처리, 재고 확인 등 일상적인 업무는 모두 현재 데이터를 사용한다. 이력 조회는 문제가 발생했거나, 감사 요청이 있거나, 특별한 분석이 필요할 때만 한다.

그런데 현재 테이블에 이력이 함께 있으면 99%의 일반 조회에서도 이력 데이터를 함께 스캔해야 한다. 이것은 매우 비효율적이다.

테이블 분리 설계

현재 테이블과 이력 테이블을 분리하자.

```
-- 현재 데이터 테이블
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
```

```
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL
);

-- 이력 테이블
DROP TABLE IF EXISTS product_history;

CREATE TABLE product_history (
    history_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    created_at DATETIME NOT NULL,
    created_by VARCHAR(100) NOT NULL,
    -- 이력 관리 컬럼
    history_created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    history_created_by VARCHAR(100) NOT NULL,
    change_type VARCHAR(50),
    change_reason VARCHAR(500),

    INDEX idx_product_id (product_id),
    INDEX idx_history_created_at (history_created_at)
);
```

현재 테이블(`product`)은 최신 데이터만 보관한다. 이력 테이블(`product_history`)은 모든 변경 이력을 저장한다.

이력 테이블(product_history)의 두 가지 시간

이력 테이블을 보면 `created_at`과 `history_created_at`이라는 두 가지의 시간 정보를 보관한다.
이력 테이블의 `created_at`과 `history_created_at`은 의미가 완전히 다르며, 둘 다 있어야 한다.
`created_at`은 "상품(Product)이 처음 태어난 시간"이고, `history_created_at`은 "변경사항이 기록(스냅샷)된 시간"이다.

구분	created_at (원본 데이터 유지)	history_created_at (이력 시점)
의미	원본 테이블(product)의 `created_at` 값을 그대로 복사	이 이력 데이터가 `product_history` 테이블에 **INSERT 되는 순간**의 시간
질문 예시	"이 상품은 최초에 언제 등록되었나요?"	"상품의 가격이 언제 변경되었나요?" "과거 특정 시점의 데이터는 무엇이었나요?"
값의 변화	동일한 `product_id`를 가진 이력 데이터들 사이에서는 **값이 변하지 않는다.** (상품의 생일은 바뀌지 않으므로)	변경이 발생할 때마다 **매번 새로운 시간**이 기록된다.
비고	데이터의 **'속성'**에 해당한다.	데이터 변경의 **'타임라인'** 역할을 한다.

데이터 등록

상품을 등록할 때 현재 테이블과 이력 테이블 모두에 데이터를 넣는다.

```
-- 현재 테이블에 등록
INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by, created_at) VALUES ('스마트폰 케이스', 15000, 100, 'ACTIVE',
'admin_kim', 'admin_kim', '2026-01-15 10:00:00');

-- 이력 테이블에도 등록 (첫 데이터)
INSERT INTO product_history (product_id, name, price, stock_quantity, status,
created_at, created_by, history_created_at, history_created_by, change_type,
change_reason)
SELECT product_id, name, price, stock_quantity, status, created_at,
created_by, '2026-01-15 10:00:00', 'admin_kim', 'CREATE', '신규 상품 등록'
FROM product WHERE product_id = LAST_INSERT_ID();
```

- **참고**: 이후 진행할 예제를 위해 `created_at`에 날짜 데이터를 직접 입력했다.
- 이력 테이블은 현재 테이블의 데이터를 복사해서 저장한다. 이때 현재 테이블의 `created_at`도 이력 테이블의 `created_at`으로 그대로 복사한다.

두 번째 상품도 등록하자.

```
INSERT INTO product (name, price, stock_quantity, status, created_by,
```

```
updated_by, created_at)
VALUES ('무선 이어폰', 89000, 50, 'ACTIVE', 'admin_lee', 'admin_lee', '2026-01-15
10:05:00');

INSERT INTO product_history (product_id, name, price, stock_quantity, status,
created_at, created_by, history_created_at, history_created_by, change_type,
change_reason)
SELECT product_id, name, price, stock_quantity, status, created_at,
created_by, '2026-01-15 10:05:00', 'admin_lee', 'CREATE', '신규 상품 등록'
FROM product WHERE product_id = LAST_INSERT_ID();
```

현재 테이블을 확인해보자.

```
SELECT product_id, name, price, stock_quantity, created_at
FROM product;
```

[실행 결과]

product_id	name	price	stock_quantity	created_at
1	스마트폰 케이스	15000	100	2026-01-15 10:00:00
2	무선 이어폰	89000	50	2026-01-15 10:05:00

이력 테이블을 확인해보자.

```
SELECT history_id, product_id, name, price, change_type, history_created_at
FROM product_history;
```

[실행 결과]

history_id	product_id	name	price	change_type	history_created_at
1	1	스마트폰 케이스	15000	CREATE	2026-01-15 10:00:00

2	2	무선 이어폰	89000	CREATE	2026-01-15 10:05:00

- 현재 테이블과 이력 테이블 모두에 같은 데이터가 입력된 것을 확인할 수 있다.

> **이력 테이블과 현재 테이블에 최초 데이터를 함께 보관하는 이유**
> 바로 뒤에서 설명한다.

데이터 변경

스마트폰 케이스의 가격은 15000 ➜ 12000으로 변경해보자.

가격을 변경할 때는 다음과 같이 한다.

1. 이력 테이블에 새로운 상태를 INSERT
2. 현재 테이블을 UPDATE

```
-- 1. 이력 테이블에 변경 후 상태 저장
INSERT INTO product_history (product_id, name, price, stock_quantity, status,
created_at, created_by, history_created_at, history_created_by, change_type,
change_reason)
VALUES (1, '스마트폰 케이스', 12000, 100, 'ACTIVE', '2026-01-15 10:00:00',
'admin_kim', '2026-03-01 10:00:00', 'admin_park', 'PRICE_CHANGE', '봄맞이 할인 이
벤트');

-- 2. 현재 테이블 업데이트
UPDATE product
SET price = 12000,
    updated_at = '2026-03-01 10:00:00',
    updated_by = 'admin_park'
WHERE product_id = 1;
```

- `product` 테이블의 `updated_at`, `product_history` 테이블의 `history_created_at` 컬럼은 현재 날짜(`now()`)를 사용해야 하지만, 여기서는 원할한 예제 진행을 위해 날짜를 직접 지정했다.

현재 테이블 확인

```
```

```sql
SELECT product_id, name, price, updated_at, updated_by
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	updated_at	updated_by
1	스마트폰 케이스	12000	2026-03-01 10:00:00	admin_park

이력 테이블 확인

```sql
SELECT history_id, product_id, name, price, change_type, change_reason,
history_created_at
FROM product_history
WHERE product_id = 1
ORDER BY history_id;
```

[실행 결과]

history_id	product_id	name	price	change_type	change_reason	history_created_
1	1	스마트폰 케이스	15000	CREATE	신규 상품 등록	2026-01-1 10:00:00
3	1	스마트폰 케이스	12000	PRICE_CHANGE	봄맞이 할인 이벤트	2026-03-0 10:00:00

- 이력 테이블의 `history_created_at` 컬럼을 통해 이 데이터가 변경된 날짜를 확인할 수 있다.

한 번 더 변경

이번에는 재고를 100 ➜ 95로 조정하면서 한번 더 변경해보자.

```sql
-- 재고 조정
INSERT INTO product_history (product_id, name, price, stock_quantity, status,
created_at, created_by, history_created_at, history_created_by, change_type,
```

```
change_reason)
VALUES (1, '스마트폰 케이스', 12000, 95, 'ACTIVE', '2026-01-15 10:00:00',
'admin_kim', '2026-03-15 14:00:00', 'warehouse_kim', 'STOCK_ADJUST', '재고 실사 -
5개 파손');

UPDATE product
SET stock_quantity = 95,
    updated_at = NOW(),
    updated_by = 'warehouse_kim'
WHERE product_id = 1;
```

```
SELECT history_id, product_id, price, stock_quantity, change_type,
change_reason, history_created_at
FROM product_history
WHERE product_id = 1
ORDER BY history_id;
```

[실행 결과]

history_id	product_id	price	stock_quantity	change_type	change_reason	history_created_a
1	1	15000	100	CREATE	신규 상품 등록	2026-01-1 10:00:00
3	1	12000	100	PRICE_CHANGE	봄맞이 할인 이벤트	2026-03-0 10:00:00
4	1	12000	95	STOCK_ADJUST	재고 실사 - 5개 파손	2026-03-1 14:00:00

이력 테이블을 확인해보면 모든 변경 이력이 보관되고 있다.

장점

장점 1: 현재 데이터 조회 성능이 좋다

```
```

```sql
-- 가장 많이 사용하는 조회
SELECT * FROM product WHERE status = 'ACTIVE';
```

현재 테이블에는 현재 데이터만 있으므로 조회가 빠르다.

장점 2: 이력 조회도 가능하다

```sql
-- 필요할 때 이력 조회
SELECT * FROM product_history
WHERE product_id = 1
ORDER BY history_id;
```

장점 3: 테이블 관리가 용이하다

현재 테이블과 이력 테이블을 별도로 관리할 수 있다. 이력 테이블은 파티셔닝, 아카이빙 등을 적용하기 쉽다.

전체 행 스냅샷 이력 테이블 - 주의점

주의! - 이력 테이블에는 첫 데이터부터 저장해야 한다

많은 개발자들이 이력 테이블을 처음 설계할 때 "최신 데이터는 `product` 테이블에만 저장하고, **데이터가 변경되는 시점부터** 기존 데이터를 `product_history`로 옮기면 중복을 제거하고 저장 공간을 아낄 수 있지 않을까?"라고 생각한다.

언뜻 보면 합리적인 생각이다. 하지만 이 방식은 데이터를 조회할 때 **지옥 같은 복잡함**을 선물한다. 왜 그런지 실제 예제 코드로 직접 확인해보자.

1. 잘못된 설계 실험: 변경 시점에만 이력을 저장하는 경우

먼저 잘못된 방식으로 테이블을 만들고 데이터를 넣어보자. 비교를 위해 `_bad` 라는 접미사를 붙여 테이블을 생성한다.

```sql
DROP TABLE IF EXISTS product_bad;
DROP TABLE IF EXISTS product_history_bad;
```

```sql
-- 잘못된 설계: 현재 테이블
CREATE TABLE product_bad (
    product_id BIGINT PRIMARY KEY,
    price INT,
    updated_at DATETIME
);

-- 잘못된 설계: 이력 테이블 (변경 전 데이터를 저장)
CREATE TABLE product_history_bad (
    history_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT,
    price INT,
    updated_at DATETIME
);
```

- 예제를 단순화 하기 위해 날짜는 `updated_at` 컬럼 하나만 사용했다.

상황 1: 상품 등록 (INSERT)

상품을 10,000원에 등록한다. 잘못된 설계에서는 `INSERT` 시점에 이력을 남기지 않는다.

```sql
INSERT INTO product_bad (product_id, price, updated_at) VALUES (1, 10000,
'2026-01-01 10:00:00');
```

이 시점에 이력 테이블을 조회해보자.

```sql
SELECT * FROM product_history_bad;
```

[실행 결과]

history_id	product_id	price	updated_at
(NULL)	(NULL)	(NULL)	(NULL)

문제점: 이 상품이 언제 처음 생성되었는지, 최초의 가격은 얼마였는지 **이력 테이블만 봐서는 알 수 없다.** 즉, 이력의 끊김이 발생한다. 이 경우 원본 테이블을 추가로 확인해야 한다.

원본 테이블 조회

```
SELECT * FROM product_bad;
```

[실행 결과]

product_id	price	updated_at
1	10000	2026-01-01 10:00:00

상황 2: 가격 변경 (UPDATE)

가격을 10,000원에서 20,000원으로 인상한다. 이때 기존 데이터(10,000원)를 이력 테이블로 옮긴다.

```
-- 1. 변경 전 데이터를 이력에 저장
INSERT INTO product_history_bad (product_id, price, updated_at)
SELECT product_id, price, updated_at FROM product_bad WHERE product_id = 1;

-- 2. 현재 테이블 업데이트
UPDATE product_bad
SET price = 20000, updated_at = '2026-02-01 10:00:00'
WHERE product_id = 1;
```

원본 테이블을 조회

```
SELECT * FROM product_bad;
```

[실행 결과]

product_id	price	updated_at
1	20000	2026-02-01 10:00:00

이력 테이블 조회

```
SELECT * FROM product_history_bad;
```

[실행 결과]

history_id	product_id	price	updated_at
1	1	10000	2026-01-01 10:00:00

원본 테이블에는 20000원과 2026-02-01 날짜가, 이력 테이블에는 과거의 이력인 10000원과 2026-01-01 날짜가 저장된 것을 확인할 수 있다. 여기까지는 문제가 없어 보인다.

상황 3: 과거 내역 조회 (문제의 발생)
이제 현업 부서에서 **"이 상품의 전체 가격 변동 내역을 시간순으로 뽑아주세요"** 라고 요청했다.

이력 테이블만 조회하면 될까?

```
SELECT * FROM product_history_bad;
```

[실행 결과]

history_id	product_id	price	updated_at
1	1	10000	2026-01-01 10:00:00

- 현재 가격인 `20000`원은 당연히 나오지 않는다. 이력 테이블에는 '과거'만 있고 '현재'가 없기 때문이다.
- 완벽한 이력을 보려면 **현재 테이블과 이력 테이블을 합쳐야(UNION) 한다.**

```
-- 잘못된 설계에서 전체 이력을 조회하는 쿼리
SELECT product_id, price, updated_at FROM product_bad WHERE product_id = 1
UNION ALL
SELECT product_id, price, updated_at FROM product_history_bad WHERE product_id
= 1
ORDER BY updated_at;
```

[실행 결과]

product_id	price	updated_at
1	10000	2026-01-01 10:00:00
1	20000	2026-02-01 10:00:00

단순한 이력 조회를 위해 매번 두 테이블을 `UNION` 해야 한다. 만약 테이블에 컬럼이 50개라면 쿼리는 끔찍하게 길어질 것이다. 또한 "특정 시점(예: 1월 15일)의 가격"을 조회하려고 하면 로직은 훨씬 더 복잡해진다.

2. 올바른 설계 확인: 처음부터 다 저장하는 경우

우리가 앞서 설계한 `product`와 `product_history` 테이블의 방식을 보자. 우리는 `INSERT` 시점부터 이력을 저장했다.

앞선 예제에서 이미 데이터가 들어가 있으므로 조회만 다시 해보자.

```sql
-- 올바른 설계: 오직 이력 테이블만 조회하면 된다.
SELECT price, history_created_at, change_type
FROM product_history
WHERE product_id = 1
ORDER BY history_created_at;
```

[실행 결과]

price	history_created_at	change_type
15000	2026-01-15 10:00:00	CREATE
12000	2026-03-01 10:00:00	PRICE_CHANGE
12000	2026-03-15 14:00:00	STOCK_ADJUST

현재 테이블(`product`)을 전혀 신경 쓰지 않고도, 상품의 탄생부터 현재까지의 모든 역사가 완벽하게 조회된다. 쿼리도 단순하고, 개발자의 실수 가능성도 거의 없다.

실무에서의 문제점

데이터를 변경 시점에만 저장하면 다음과 같은 실무적인 문제가 발생한다.

- **쿼리의 복잡도 증가**: 과거 시점의 데이터를 완벽하게 복원하기 위해서는 항상 `현재 테이블 + 이력 테이블`을 조합해서 봐야 한다. 쿼리가 2배 이상 길어지고 버그가 발생할 확률이 높아진다.
- **이력의 불연속성**: `INSERT` 이벤트가 이력 테이블에 없기 때문에, "이 상품이 언제 태어났는지"에 대한 기록이 이력 테이블에서 누락된다. 이력 테이블은 그 자체로 **완결된 타임라인**을 가져야 가치가 있다.
- **개발 생산성 저하**: 저장 공간을 조금 아끼려다가, 데이터를 조회하고 분석하는 모든 개발자가 복잡한 로직을 매번 구현해야 한다.
- **원본 테이블 복원 불가**: 누군가 실수로 이력 테이블에 데이터를 보관하는 것을 깜박하고, 원본 테이블의 데이터만 삭제하면 원본 테이블의 데이터를 복원할 수 없다.

결론: 첫 데이터부터 무조건 저장해라

결론은 명확하다. **처음 `INSERT` 하는 순간에도 이력 테이블에 데이터를 똑같이 넣어라.**

```
-- 올바른 패턴: 등록(INSERT) 시점에도 이력을 남긴다.
INSERT INTO product (...) VALUES (...);
INSERT INTO product_history (...) VALUES (..., 'CREATE', '신규 등록');
```

이렇게 하면 다음과 같은 장점을 얻는다.

1. **단순한 조회**: 과거 내역이나 특정 시점의 데이터가 필요하면 **오직 `product_history` 테이블만 조회하면 된다.** 현재 테이블을 신경 쓸 필요가 없다.
2. **완벽한 타임라인**: 생성(`CREATE`)부터 수정(`UPDATE`), 삭제(`DELETE`)까지 모든 생애 주기가 이력 테이블 하나에 온전히 담긴다.
3. **유지보수성**: 디스크 용량은 저렴하지만, 개발자의 시간과 데이터 정합성을 맞추는 비용은 비싸다. 단순한 구조가 최고의 설계다.
4. **원본 테이블 복원 가능**: 이력 테이블에 처음부터 원본 데이터가 저장되어 있다. 따라서 누군가 실수로 원본 테이블의 데이터를 삭제해도 이력 테이블을 통해 저장된 원본 데이터를 확인하고 복원할 수 있다.

저장 공간을 조금 아끼려다 개발 생산성을 포기하지 마라. 디스크 용량은 싸지만, 개발자의 시간과 데이터 정합성을 맞추는 비용은 매우 비싸다.

- **잘못된 패턴**: `INSERT` 생략, `UPDATE`, `DELETE` 시에만 이력 저장 ➜ 조회할 때 `UNION` 지옥을 맛봄.

- **올바른 패턴**: `INSERT` 포함, 모든 변경 시점에 이력 저장 ➜ `SELECT` 한 번으로 끝남.

무조건 **첫 데이터(INSERT)부터 이력 테이블에 저장해라.** 이것이 실무의 정석이다.

다음 수업에서는 이력 테이블에 유효 기간을 추가하는 방법과 실무에서의 트레이드오프에 대해 알아보겠다.

전체 행 스냅샷 이력 테이블 - 유효 기간

이전 수업에서 현재 테이블과 이력 테이블을 분리했다.
이력 테이블에서 특정 시점의 데이터를 조회하려면 어떻게 해야 할까?

> **하나의 테이블로 이력 관리의 내용을 떠올려보자.**
> 우리는 앞서 현재 테이블로 이력 관리 - 유효기간에서 이미 이런 해결 방식을 살펴보았다.
> 이번에는 같은 내용을 현재와 이력이 분리된 테이블에 적용하는 것이다.
> 내용을 복습한다고 생각하고 빠르게 살펴보겠다.

시점 조회의 문제

이력 테이블 구조에서 특정 상품의 "2026년 3월 10일 기준 가격"을 조회해보자.
참고로 이력 테이블에는 과거 이력 뿐만 아니라 현재 데이터도 함께 존재한다. 따라서 이력 테이블만 조회해도 필요한 모든 데이터를 찾을 수 있다.

```sql
SELECT product_id, name, price, history_created_at
FROM product_history
WHERE product_id = 1
  AND history_created_at <= '2026-03-10 23:59:59'
ORDER BY history_created_at DESC
LIMIT 1;
```

[실행 결과]

product_id	name	price	history_created_at

1	스마트폰 케이스	12000	2026-03-01 10:00:00

단일 상품 조회는 문제없다. 하지만 전체 상품의 특정 시점 통계를 내려면 복잡한 서브쿼리가 필요하다.

```
-- 2026년 3월 10일 기준 전체 재고
SELECT SUM(h.stock_quantity) AS total_stock
FROM product_history h
INNER JOIN (
    SELECT product_id, MAX(history_created_at) AS max_history_at
    FROM product_history
    WHERE history_created_at <= '2026-03-10 23:59:59'
    GROUP BY product_id
) latest ON h.product_id = latest.product_id
        AND h.history_created_at = latest.max_history_at;
```

[실행 결과]

total_stock
150

우리는 이 문제를 이미 앞서 테이블 하나로 이력 관리를 할 때 살펴보았다.
이 문제는 앞서 배운 유효 기간 `valid_from`, `valid_to`를 적용하면 해결할 수 있다.
이력 테이블에는 모든 데이터가 다 저장되어 있으므로, 여기에 유효 기간만 추가하고 관리하면 된다.

유효 기간이 추가된 이력 테이블

```
DROP TABLE IF EXISTS product_history;

CREATE TABLE product_history (
    history_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
```

```
    -- 유효 기간
    valid_from DATETIME NOT NULL,
    valid_to DATETIME NOT NULL DEFAULT '9999-12-31 23:59:59',
    -- 기타 컬럼
    created_at DATETIME NOT NULL,
    created_by VARCHAR(100) NOT NULL,
    history_created_by VARCHAR(100) NOT NULL,
    change_type VARCHAR(50),
    change_reason VARCHAR(500),

    INDEX idx_product_id_valid_range (product_id, valid_from, valid_to),
    INDEX idx_valid_range (valid_from, valid_to)
);
```

- 현재 데이터를 조회할 때는 원본 테이블을 직접 확인하면 된다. 따라서 이렇게 원본 테이블과 이력 테이블이 나누어진 경우에는 `is_current` 컬럼을 사용할 필요가 없다.

데이터 등록 및 변경

데이터를 등록하고 두 번 변경해보자.

```
-- 1. 최초 등록 (2026-01-15)
INSERT INTO product_history (product_id, name, price, stock_quantity, status,
valid_from, valid_to, created_at, created_by, history_created_by, change_type,
change_reason)
VALUES (1, '스마트폰 케이스', 15000, 100, 'ACTIVE', '2026-01-15 10:00:00',
'9999-12-31 23:59:59', '2026-01-15 10:00:00', 'admin_kim', 'admin_kim',
'CREATE', '신규 상품 등록');

INSERT INTO product_history (product_id, name, price, stock_quantity, status,
valid_from, valid_to, created_at, created_by, history_created_by, change_type,
change_reason)
VALUES (2, '무선 이어폰', 89000, 50, 'ACTIVE', '2026-01-15 10:05:00', '9999-12-31
23:59:59', '2026-01-15 10:05:00', 'admin_lee', 'admin_lee', 'CREATE', '신규 상품
등록');
```

- 원칙적으로 `product` 테이블에 데이터를 입력하고 또 변경해야 하지만, 여기서는 예제의 단순함을 위해 `product_history` 테이블만 데이터를 입력하고 변경하겠다.

[저장된 데이터]

history_id	product_id	name	price	stock_quantity	valid_from	valid_to	ch_ty
1	1	스마트폰 케이스	15000	100	2026-01-15 10:00:00	9999-12-31 23:59:59	CR
2	2	무선 이어폰	89000	50	2026-01-15 10:05:00	9999-12-31 23:59:59	CR

- 최초 입력된 데이터들은 `valid_to`가 `9999-12-31 23:59:59`인 것을 확인할 수 있다. 이 값을 통해 해당 행이 현재 값인 것도 확인할 수 있다.

```
-- 2. 첫 번째 변경 (2026-03-01): 가격 변경
-- 이전 행의 valid_to 업데이트
UPDATE product_history
SET valid_to = '2026-03-01 10:00:00'
WHERE product_id = 1 AND valid_to = '9999-12-31 23:59:59';

-- 새 행 추가
INSERT INTO product_history (product_id, name, price, stock_quantity, status,
valid_from, valid_to, created_at, created_by, history_created_by, change_type,
change_reason)
VALUES (1, '스마트폰 케이스', 12000, 100, 'ACTIVE', '2026-03-01 10:00:00',
'9999-12-31 23:59:59', '2026-01-15 10:00:00', 'admin_kim', 'admin_park',
'PRICE_CHANGE', '봄맞이 할인');
```

- `valid_to`를 데이터 변경 시점에 맞추어 변경해야 한다.

[저장된 데이터]

history_id	product_id	name	price	stock_quantity	valid_from	valid_to	ch_ty
1	1	스마트폰 케이스	15000	100	2026-01-15 10:00:00	2026-03-01 10:00:00	CR
2	1	무선 이어폰	89000	50	2026-01-15 10:05:00	9999-12-31 23:59:59	CR

3	1	스마트폰 케이스	12000	100	2026-03-01 10:00:00	9999-12-31 23:59:59	PR _C

- `history_id=1`의 `valid_to`가 `9999-12-31 23:59:59`가 변경 시점인 `2026-03-01 10:00:00`으로 변경되었다.

```
-- 3. 두 번째 변경 (2026-03-15): 재고 조정
UPDATE product_history
SET valid_to = '2026-03-15 14:00:00'
WHERE product_id = 1 AND valid_to = '9999-12-31 23:59:59';

INSERT INTO product_history (product_id, name, price, stock_quantity, status,
valid_from, valid_to, created_at, created_by, history_created_by, change_type,
change_reason)
VALUES (1, '스마트폰 케이스', 12000, 95, 'ACTIVE', '2026-03-15 14:00:00',
'9999-12-31 23:59:59', '2026-01-15 10:00:00', 'admin_kim', 'warehouse_kim',
'STOCK_ADJUST', '재고 실사');
```

- 현재 데이터를 변경할 때는 반드시 이력 테이블의 `valid_to` 값도 변경 시점에 맞추어 함께 변경해야 한다.

최종 이력을 확인해보자.

```
SELECT history_id, product_id, price, stock_quantity, valid_from, valid_to,
change_type
FROM product_history
ORDER BY valid_from;
```

[실행 결과]

history_id	product_id	price	stock _quantity	valid_from	valid_to	change_
1	1	15000	100	2026-01-15 10:00:00	2026-03-01 10:00:00	CREATE
2	2	89000	50	2026-01-15 10:05:00	9999-12-31 23:59:59	CREATE

3	1	12000	100	2026-03-01 10:00:00	2026-03-15 14:00:00	PRICE_C E
4	1	12000	95	2026-03-15 14:00:00	9999-12-31 23:59:59	STOCK_ T

- `valid_to`가 변경 시점에 맞추어 적절하게 변경된 것을 확인할 수 있다.
- `valid_to`의 값이 `9999-12-31 23:59:59`이면 현재 데이터이다.

시점 조회 - 단순해진 쿼리

특정 상품 하나를 조회해보자.

```
-- 2026년 3월 10일 기준 가격
SELECT product_id, name, price, stock_quantity, valid_from, valid_to
FROM product_history
WHERE product_id = 1
  AND '2026-03-10 23:59:59' >= valid_from
  AND '2026-03-10 23:59:59' < valid_to;
```

[실행 결과]

product_id	name	price	stock_quantity	valid_from	valid_to
1	스마트폰 케이스	12000	100	2026-03-01 10:00:00	2026-03-15 14:00:00

전체 통계 - 성능 개선

"2026년 3월 10일 기준, 모든 상품의 총 재고 수량"을 조회해보자.

```
SELECT SUM(h.stock_quantity) AS total_stock
FROM product_history h
WHERE '2026-03-10 23:59:59' >= h.valid_from
  AND '2026-03-10 23:59:59' < h.valid_to;
```

[실행 결과]

total_stock
150

이 경우에도 과거와는 비교할 수 없을 정도로 쿼리가 매우 단순해졌다. 서브쿼리나 조인 없이 단순 WHERE 조건만으로 해결된다.
또한 범위 컬럼에 인덱스를 사용해서 성능도 이전과 비교할 수 없을 정도로 빠르다.

트레이드오프 분석

이력 테이블에 유효 기간(`valid_from`, `valid_to`)을 사용하면 시점 조회가 편리해진다. 하지만 장단점이 있다.

장점

1. **시점 조회가 단순하다**: 범위 조건만으로 특정 시점 데이터를 조회할 수 있다.
2. **통계 쿼리 성능이 좋다**: 서브쿼리 없이 집계할 수 있다.
3. **인덱스 활용이 좋다**: `product_id, valid_from, valid_to` 인덱스로 효율적인 검색이 가능하다.

단점

1. **이력 테이블이 불변하지 않다**: 새 데이터가 추가될 때마다 이전 행의 `valid_to`를 UPDATE해야 한다.
2. **트랜잭션 복잡도 증가**: UPDATE와 INSERT가 원자적으로 실행되어야 한다.
3. **장애 시 데이터 정합성 문제**: UPDATE 후 INSERT 전에 장애가 발생하면 데이터가 깨질 수 있다. (트랜잭션 필요)

트랜잭션 운영 예시

```
-- 실무에서 변경 처리
START TRANSACTION;

-- 이전 행의 유효 기간 종료
UPDATE product_history
SET valid_to = NOW()
WHERE product_id = 1 AND valid_to = '9999-12-31 23:59:59';
```

```sql
-- 새 행 추가
INSERT INTO product_history (...)
VALUES (...);

-- 현재 테이블도 업데이트
UPDATE product SET ... WHERE product_id = 1;

COMMIT;
```

유효 기간 없이 운영하는 방법

물론 유효 기간 없이도 시점 조회는 가능하다. 다만 통계 쿼리가 복잡하고 성능이 느릴 뿐이다.

```sql
-- valid_from, valid_to 없이 시점 조회
SELECT h.*
FROM product_history h
INNER JOIN (
    SELECT product_id, MAX(history_id) AS max_id
    FROM product_history
    WHERE history_created_at <= '2026-03-10 23:59:59'
    GROUP BY product_id
) latest ON h.history_id = latest.max_id
WHERE h.product_id = 1;
```

실무 권장 사항

실무에서는 요구사항에 기간 조회가 필요하다면 유효 기간 컬럼을 추가하는 것이 좋다.

선택 예시

상황	권장 방식
시점 조회가 거의 없다	유효 기간 없이 운영
시점 조회가 자주 필요하다	유효 기간 추가
시점 기반 통계 배치가 많다	유효 기간 추가

이력 테이블 불변성이 중요하다	유효 기간 없이 운영

실무 예시

1. 유효 기간 컬럼이 필요한 테이블 (Master Data)

주로 데이터의 **'상태'**가 일정 기간 동안 유지되는 마스터성 정보에 적합하다.

- **대상**: `product`(상품), `member`(회원 등급), `category`(카테고리), `department`(부서) 등
- **특징**: 데이터가 생성되면 그 상태가 일정 기간 지속된다. 변경 빈도가 트랜잭션 데이터에 비해 상대적으로 낮다.
- **이유**: "2024년 12월 25일 당시의 상품 가격은 얼마였는가?", "작년 이벤트 기간 동안 회원의 등급은 무엇이었는가?"와 같이 과거의 상태를 조회해야 하는 비즈니스 요구사항이 빈번하다.

2. 유효 기간 컬럼이 불필요한 테이블 (Transaction/Log Data)

특정 시점에 발생한 **'사건(Event)'**을 기록하는 테이블에는 적합하지 않다.

- **대상**: `orders`(주문), `payments`(결제), `point_history`(포인트 내역), `access_log`(접속 로그) 등
- **특징**: 데이터가 발생하는 그 순간의 시간이 중요하며, 한 번 발생하면 변하지 않는 불변(Immutable)의 성격을 가진다. 데이터가 쌓이는 속도가 매우 빠르고 양이 방대하다.
- **이유**: 주문이나 결제는 '기간'의 개념이 아니라 '발생 시점(`created_at`)'이 핵심이다. "주문이 1시부터 2시까지 유효했다"라는 개념은 어색하다. 또한 데이터 양이 막대한 테이블에서 이력을 남기기 위해 이전 행을 `UPDATE` 하는 방식은 성능에 영향을 줄 수 있다. 이런 데이터는 단순히 `INSERT`만 하는 방식으로 이력을 쌓는 것이 좋다.

요약 정리

구분	성격	예시 테이블	유효 기간 컬럼 사용	이유
마스터 데이터	상태 유지, 기준 정보	상품, 회원, 조직	**권장**	특정 시점의 상태(가격, 등급 등) 조회가 중요함

트랜잭션 데이터	사건 발생, 이력 기록	주문, 결제, 로그	**비권장**	발생 시점이 중요하며, 데이터 양이 많아 UPDATE 비용이 큼

실무에서는 이 기준을 바탕으로, 비즈니스 요건에 따라 시점 조회가 꼭 필요한 테이블에만 선별적으로 유효 기간(`valid_from`, `valid_to`)을 도입하는 것이 시스템 복잡도를 낮추는 길이다.

다음 수업에서는 전체 행 스냅샷 방식의 한계점에 대해 알아보겠다.

전체 행 스냅샷 이력 테이블 - 한계

전체 행 스냅샷 방식은 실무에서 가장 많이 사용하는 패턴이다. 하지만 이 방식에도 분명한 한계가 있다.

한계 예시

한계 1: 용량 문제

전체 행 스냅샷은 변경이 발생할 때마다 모든 컬럼의 값을 저장한다. 하나의 컬럼만 변경되어도 전체 행이 복사된다.

예를 들어 상품 테이블에 20개의 컬럼이 있다고 가정하자. 가격 하나만 변경해도 20개 컬럼 전체가 이력 테이블에 저장된다.

```
-- 상품 테이블 예시 (실제로는 더 많은 컬럼이 있을 수 있다)
CREATE TABLE product_full (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(200),
    description TEXT,           -- 긴 텍스트
    price INT,
    original_price INT,
    discount_rate DECIMAL(5,2),
    stock_quantity INT,
    reserved_quantity INT,
    category_id BIGINT,
```

```
    brand_id BIGINT,
    supplier_id BIGINT,
    weight DECIMAL(10,2),
    dimensions VARCHAR(100),
    status VARCHAR(20),
    display_order INT,
    created_at DATETIME,
    created_by VARCHAR(100),
    updated_at DATETIME,
    updated_by VARCHAR(100)
);
```

- 설명을 위한 예시이다. 실행하지 말자.

가격만 변경해도 `description`(긴 텍스트) 등 모든 데이터가 복사된다.

용량 계산 예시

- 1개 상품 평균 크기: 약 2KB
- 1개 상품 평균 변경 횟수: 연간 50회
- 상품 수: 10만 개

연간 이력 테이블 증가량

- = 2KB × 50회 × 100,000개
- = 10GB / 년

5년 운영 시

- = 50GB 이상의 이력 데이터

한계 2: 불필요한 중복

가격만 변경되었는데 `name` 등 변경되지 않은 값들도 모두 중복 저장된다.

```
-- 이력 테이블 조회
SELECT history_id, product_id, name, price
FROM product_history
WHERE product_id = 1
ORDER BY history_id;
```

[실행 결과]

history_id	product_id	name	price
1	1	스마트폰 케이스	15000
3	1	스마트폰 케이스	12000
4	1	스마트폰 케이스	10000

- `name`을 포함한 변경하지 않은 나머지 필드들이 중복으로 저장된다.

한계 3: 무엇이 변경되었는지 알기 어렵다

전체 행이 저장되므로 "무엇이 변경되었는지"를 파악하려면 이전 행과 비교해야 한다. 쿼리가 복잡하고, 컬럼이 많아질수록 더 복잡해진다.

그럼에도 전체 행 스냅샷을 사용하는 이유

이런 한계에도 불구하고 실무에서 전체 행 스냅샷 방식이 가장 많이 사용되는 이유가 있다.

이유 1: 복원이 쉽다

특정 시점의 전체 상태를 그대로 확인할 수 있다. 문제가 발생했을 때 "그때 데이터가 정확히 어땠는지" 바로 알 수 있다.

```
-- 특정 시점의 전체 데이터를 그대로 확인
SELECT * FROM product_history WHERE history_id = 3;
```

이유 2: 쿼리가 단순하다

이력 조회, 시점 조회 등이 직관적이다.

```
-- 이력 조회가 단순하다
SELECT * FROM product_history WHERE product_id = 1;
```

이유 3: 다양한 분석이 가능하다

전체 데이터가 있으므로 어떤 분석이든 가능하다.

```sql
-- 2026년 3월 10일 기준, 모든 상품의 총 재고 수량
SELECT SUM(h.stock_quantity) AS total_stock
FROM product_history h
WHERE '2026-03-10 23:59:59' >= h.valid_from
  AND '2026-03-10 23:59:59' < h.valid_to;
```

```sql
-- 변경 사유별 카운트
SELECT change_type, count(*)
FROM product_history
GROUP BY change_type;
```

이유 4: 저장 비용이 저렴해졌다
과거에는 저장 용량이 비쌌지만, 현재는 저장 비용이 매우 저렴하다. 용량보다 개발 편의성과 운영 안정성이 더 중요한 경우가 많다.

정리: 그래서 언제 써야 할까?

전체 행 스냅샷 방식은 **'저장 용량'을 희생하고 '개발 생산성'과 '운영 안정성'을 얻는 전략**이다.

실무에서는 특별한 이유가 없다면 이 방식을 **기본(Default)**으로 선택하는 것을 권장한다. 최근의 하드웨어 기술 발전으로 인해 저장 공간의 비용은 계속 낮아지고 있다. 반면, 복잡한 이력 관리 로직을 구현하고 유지보수하는 인건비와, 잘못된 데이터 조회로 인한 리스크 비용은 훨씬 크다.

핵심 요약

- **장점**: 구현이 쉽고, 조회 쿼리가 단순하며, 특정 시점의 데이터 복원이 완벽하다.
- **단점**: 저장 공간을 많이 차지하고, 불필요한 중복 데이터가 발생한다.
- **결론**: 데이터 양이 천문학적인 수준이 아니라면, 전체 행 스냅샷이 가장 합리적인 선택이다.

하지만 데이터 변경이 초당 수천 건씩 발생하거나, 하나의 테이블에 컬럼이 수백 개가 넘어가서 전체를 복사하는 비용이 감당되지 않는 상황이라면 어떻게 해야 할까? 이때는 변경된 부분만 기록하는 방식이 필요하다.

다음 수업에서는 전체 행 스냅샷의 대안인 컬럼 단위 변경 로그에 대해 알아보겠다.

컬럼 단위 변경 로그 테이블

전체 행 스냅샷 방식의 용량 문제를 해결하는 방법 중 하나는 "변경된 컬럼만" 저장하는 것이다. 이것을 필드 레벨 추적 (Field-Level Tracking) 또는 컬럼 단위 변경 로그라고 한다.

아이디어

가격이 변경되면 가격만 저장한다. 재고가 변경되면 재고만 저장한다. 변경되지 않은 컬럼은 저장하지 않는다.

테이블 설계

```
DROP TABLE IF EXISTS product_change_log;

CREATE TABLE product_change_log (
    log_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    column_name VARCHAR(100) NOT NULL,
    old_value VARCHAR(1000),
    new_value VARCHAR(1000),
    changed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(100) NOT NULL,
    change_reason VARCHAR(500),

    INDEX idx_product_id (product_id),
    INDEX idx_changed_at (changed_at),
    INDEX idx_column_name (column_name)
);
```

각 컬럼의 의미는 다음과 같다.

- `column_name` : 변경된 컬럼 이름
- `old_value` : 변경 전 값
- `new_value` : 변경 후 값

데이터 변경 기록

상품의 가격과 재고를 변경해보자.

```
-- 가격 변경: 15000 → 12000
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by, changed_at, change_reason) VALUES (1, 'price', '15000', '12000',
'admin_park', '2026-03-01 10:00:00', '봄맞이 할인 이벤트');

-- 재고 변경: 100 → 95
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by, changed_at, change_reason) VALUES (1, 'stock_quantity', '100',
'95', 'warehouse_kim', '2026-03-15 14:00:00', '재고 실사 - 5개 파손');

-- 상태 변경: ACTIVE → INACTIVE
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by, changed_at, change_reason) VALUES (1, 'status', 'ACTIVE',
'INACTIVE', 'admin_lee', '2026-04-01 10:00:00', '제조사 단종 통보');
```

- product 테이블의 내용도 변경했다고 가정하자.

변경 로그를 확인해보자.

```
SELECT log_id, product_id, column_name, old_value, new_value, changed_at,
changed_by
FROM product_change_log
WHERE product_id = 1
ORDER BY changed_at;
```

[실행 결과]

log_id	product_id	column_name	old_value	new_value	changed_at	char
1	1	price	15000	12000	2026-03-01 10:00:00	adm
2	1	stock_quantity	100	95	2026-03-15 14:00:00	ware
3	1	status	ACTIVE	INACTIVE	2026-04-01 10:00:00	adm

- `column_name` 컬럼을 통해 어떤 컬럼의 값이 변경되었는지 확인할 수 있다.

- `old_value`, `new_value` 필드를 통해 이전 값과 변경된 값을 확인할 수 있다.

장점

장점 1: 용량 절약

변경된 컬럼만 저장하므로 용량이 크게 줄어든다.

- 전체 행 스냅샷: 모든 컬럼 저장 (20개 컬럼 × 3번 변경 = 60개 값)
- 컬럼 단위 로그: 변경된 컬럼만 저장 (3개 값)

장점 2: 무엇이 변경되었는지 명확하다

각 로그가 "어떤 컬럼이 어떻게 변경되었는지"를 명확히 보여준다.

```
SELECT column_name, old_value, new_value, change_reason
FROM product_change_log
WHERE product_id = 1;
```

[실행 결과]

column_name	old_value	new_value	change_reason
price	15000	12000	봄맞이 할인 이벤트
stock_quantity	100	95	재고 실사 - 5개 파손
status	ACTIVE	INACTIVE	제조사 단종 통보

심각한 단점

단점 1: 특정 시점의 전체 상태를 복원하기 어렵다

"2026년 3월 20일 기준 상품 상품 ID 1의 전체 정보"를 조회하려면 어떻게 해야 할까?

```
-- 각 컬럼별로 해당 시점의 값을 찾아야 한다
-- 매우 복잡한 쿼리가 필요하다
```

```sql
-- price 값 찾기
SELECT new_value FROM product_change_log
WHERE product_id = 1 AND column_name = 'price'
  AND changed_at <= '2026-03-20'
ORDER BY changed_at DESC LIMIT 1;

-- stock_quantity 값 찾기
SELECT new_value FROM product_change_log
WHERE product_id = 1 AND column_name = 'stock_quantity'
  AND changed_at <= '2026-03-20'
ORDER BY changed_at DESC LIMIT 1;

-- status 값 찾기
SELECT new_value FROM product_change_log
WHERE product_id = 1 AND column_name = 'status'
  AND changed_at <= '2026-03-20'
ORDER BY changed_at DESC LIMIT 1;

-- 모든 컬럼에 대해 반복해야 한다...
```

[실행 결과] - price

new_value
12000

[실행 결과] - stock_quantity

new_value
95

[실행 결과] - status

new_value
null

컬럼이 20개라면 20번의 서브쿼리가 필요하다. 실용적이지 않다.

단점 2: 통계 쿼리가 매우 어렵다

"2026년 3월 20일 기준 전체 상품의 총 재고"를 구하려면?

- 각 상품별로 해당 시점의 `stock_quantity`를 찾아야 한다
- 변경 이력이 없는 상품은 원본 테이블에서 가져와야 한다
- 매우 복잡하고 성능도 나쁘다

단점 3: 타입 정보가 손실된다

모든 값이 VARCHAR로 저장되므로 원래 타입 정보가 손실된다.

```
-- price는 원래 INT지만 VARCHAR로 저장된다
-- 숫자 비교, 연산이 어려워진다
SELECT * FROM product_change_log
WHERE column_name = 'price'
  AND CAST(new_value AS SIGNED) > 10000;  -- 타입 변환 필요(MySQL은 묵시적 형변환 가능)
```

단점 4: 첫 데이터 처리가 복잡하다

로그 테이블만 있으면 과거의 어떤 시점이든 완벽하게 복구할 수 있어야 한다는 가정이 있다면 다음과 같이 복잡한 운영 방식이 요구된다. 이 경우 첫 등록 시에는 모든 컬럼의 값을 로그로 기록해야 한다. 그렇지 않으면 로그 테이블을 통해 초기 값을 알 수 없다.

```
-- 첫 등록 시 모든 컬럼을 기록해야 한다
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by)
VALUES (1, 'name', NULL, '스마트폰 케이스', 'admin_kim');
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by)
VALUES (1, 'price', NULL, '15000', 'admin_kim');
INSERT INTO product_change_log (product_id, column_name, old_value, new_value,
changed_by)
VALUES (1, 'stock_quantity', NULL, '100', 'admin_kim');
-- ... 20개 컬럼이면 20개의 INSERT
```

JSON 방식의 변형

컬럼 단위 대신 변경된 내용을 JSON으로 저장하는 방식도 있다.

```
DROP TABLE IF EXISTS product_change_log_json;

CREATE TABLE product_change_log_json (
    log_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    changes JSON NOT NULL,
    changed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(100) NOT NULL,

    INDEX idx_product_id (product_id)
);

-- 변경 내용을 JSON으로 저장
INSERT INTO product_change_log_json (product_id, changes, changed_by)
VALUES (1, '{"price": {"old": 15000, "new": 12000}, "stock_quantity": {"old":
100, "new": 95}}', 'admin_park');
```

```
SELECT log_id, product_id, changes
FROM product_change_log_json
WHERE product_id = 1;
```

[실행 결과]

log_id	product_id	changes
1	1	{"price": {"old": 15000, "new": 12000}, "stock_quantity": {"old": 100, "new": 95}}

JSON 방식은 한 번의 변경에서 여러 컬럼이 변경된 경우를 하나의 행으로 관리할 수 있다. 하지만 특정 시점 복원, 통계 쿼리 등의 문제는 여전히 남아 있다.

> **JSON에 대한 부분은 뒤에서 자세히 설명한다.**
> 지금은 JSON이라는 형식을 사용해서 데이터를 한줄에 저장하는 것도 가능하구나 정도만 알아두자.

컬럼 단위 변경 로그 정리

항목	평가
용량	매우 좋음
변경 내용 파악	매우 좋음
시점 복원	매우 어려움
통계 쿼리	매우 어려움
구현 복잡도	높음
쿼리 복잡도	높음

언제 사용하면 좋을까?

컬럼 단위 변경 로그는 다음과 같은 경우에 적합하다.

1. **특정 필드의 변경 이력만 필요한 경우**: 가격 변경 이력, 상태 변경 이력 등
2. **전체 상태 복원이 필요 없는 경우**: 변경 내역 확인만 하면 되는 경우
3. **용량이 매우 중요한 경우**: 대용량 데이터, 변경이 빈번한 데이터

실무의 결론: 전체 행 스냅샷을 우선해라

지금까지 전체 행 스냅샷 방식과 컬럼 단위 변경 로그 방식을 모두 살펴보았다. 각각의 장단점이 뚜렷하지만, 실무 경험을 바탕으로 결론을 내리자면 **대부분의 상황에서는 '전체 행 스냅샷' 방식을 선택하는 것이 정답이다.**

그 이유는 다음과 같다.

1. 개발 비용이 스토리지 비용보다 비싸다

이것이 가장 현실적인 이유다. 현대의 하드웨어 환경에서 디스크 용량은 매우 저렴하다. 반면, 복잡한 로직을 구현하고 유지보수하는 개발자의 인건비는 매우 비싸다.

- **컬럼 단위 로그**: 구현이 복잡하고, 데이터를 조회할 때마다 버그가 발생할 확률이 높다. 이를 해결하기 위해 투입되는 시간과 비용이 스토리지 절약 비용을 훨씬 상회한다.
- **전체 행 스냅샷**: `SELECT *` 한 번으로 데이터를 조회할 수 있다. 구현이 단순하여 버그가 발생할 여지가 적고 개발 속도가 빠르다.

2. '과거 상태 복원'이 이력 관리의 핵심이다

데이터 변경 이력을 남기는 가장 큰 목적은 "문제가 발생했을 때 과거의 특정 시점으로 데이터를 되돌리거나, 그 시점의 상태를 확인하기 위함"이다.

- 앞서 보았듯이 컬럼 단위 로그 방식은 특정 시점의 전체 데이터를 복원하기 위해 엄청난 비용(복잡한 쿼리, CPU 연산)을 치러야 한다.
- 반면 전체 행 스냅샷은 단순히 해당 시점의 레코드를 읽기만 하면 된다. 이력 데이터는 '저장'보다 '조회'가 더 중요한 순간(장애 대응, 감사 등)에 빛을 발해야 한다.

3. 데이터 분석의 용이성

비즈니스적으로 "지난달 1일부터 10일까지 상품 가격의 변화 추이" 혹은 "특정 기간 동안 재고가 10개 미만이었던 상품 목록" 등을 뽑아야 할 때가 있다.

- 전체 행 스냅샷은 일반적인 SQL(`GROUP BY`, `WHERE`)을 그대로 사용할 수 있어 분석이 매우 쉽다.
- 컬럼 단위 로그는 이러한 통계 쿼리를 작성하는 것이 사실상 불가능하거나, 별도의 애플리케이션 로직을 거쳐야 한다.

정리: 이력 테이블 설계 원칙

실무에서 데이터베이스 설계를 할 때는 다음 순서를 따르는 것을 권장한다.

1. **기본 전략**: 언제나 **전체 행 스냅샷(이력 테이블)** 방식을 기본으로 채택한다. 구현이 쉽고 조회가 간편하며 안전하다.
2. **예외 상황**: 데이터가 수억 건 이상 쌓여 스토리지 비용이 감당하기 힘들 정도로 커지거나, 특정 필드(예: 게시글 내용)가 너무 커서 전체를 복사하는 것이 시스템에 무리를 주는 경우에만 **제한적으로 컬럼 단위 로그**를 고려한다.
3. **타협안**: 만약 특정 필드(예: `description`)만 너무 크고 나머지는 작다면, 큰 필드만 제외하고 스냅샷을 찍거나 별도 테이블로 분리하는 방식을 사용한다.

단순함이 복잡함을 이긴다. 특별한 이유가 없다면 단순한 구조인 전체 행 스냅샷을 사용해라.

공통 이력 테이블

마지막으로 알아볼 방식은 공통 이력 테이블이다. 모든 테이블의 변경 이력을 하나의 테이블에서 관리하는 방식이다. 이렇게 하면 하나의 테이블만 관리하면 되므로 관리가 매우 편해진다.

아이디어

시스템에 상품 테이블, 주문 테이블, 회원 테이블 등 여러 테이블이 있다. 각 테이블마다 이력 테이블을 만드는 대신, 하나의 공통 이력 테이블에서 모든 변경을 추적한다.

테이블 설계

```
DROP TABLE IF EXISTS audit_log;

CREATE TABLE audit_log (
    audit_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    table_name VARCHAR(100) NOT NULL,
    record_id VARCHAR(100) NOT NULL,
    action VARCHAR(20) NOT NULL,
    old_data JSON,
    new_data JSON,
    changed_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    changed_by VARCHAR(100) NOT NULL,
    source_system VARCHAR(50),
    client_ip VARCHAR(50),
    request_id VARCHAR(100),

    INDEX idx_table_record (table_name, record_id),
    INDEX idx_changed_at (changed_at),
    INDEX idx_changed_by (changed_by)
);
```

각 컬럼의 의미는 다음과 같다.

- `table_name`: 변경이 발생한 테이블 이름
- `record_id`: 변경된 레코드의 ID (Primary Key 값)
- `action`: 변경 유형 (INSERT, UPDATE, DELETE)

- `old_data`: 변경 전 데이터 (JSON)
- `new_data`: 변경 후 데이터 (JSON)

> **JSON에 대한 부분은 뒤에서 자세히 설명한다.**
> 지금은 JSON이라는 형식을 사용해서 데이터를 한줄에 저장하는 것도 가능하구나 정도만 알아두자.

데이터 등록 로그

```
-- 상품 등록
INSERT INTO audit_log (table_name, record_id, action, old_data, new_data,
changed_by, changed_at, source_system)
VALUES (
    'product',
    '1',
    'INSERT',
    NULL,
    '{"product_id": 1, "name": "스마트폰 케이스", "price": 15000,
"stock_quantity": 100}',
    'admin_kim',
    '2026-03-15 10:00:00',
    'WEB_ADMIN'
);

-- 주문 등록
INSERT INTO audit_log (table_name, record_id, action, old_data, new_data,
changed_by, changed_at, source_system)
VALUES (
    'orders',
    '1001',
    'INSERT',
    NULL,
    '{"order_id": 1001, "member_id": 100, "total_amount": 15000, "status":
"PENDING"}',
    'member_100',
    '2026-03-15 14:30:00',
    'WEB_USER'
);

-- 회원 정보 수정
INSERT INTO audit_log (table_name, record_id, action, old_data, new_data,
```

```
changed_by, changed_at, source_system)
VALUES (
    'member',
    '100',
    'UPDATE',
    '{"member_id": 100, "name": "홍길동", "email": "hong@old.com"}',
    '{"member_id": 100, "name": "홍길동", "email": "hong@new.com"}',
    'member_100',
    '2026-03-15 15:00:00',
    'MOBILE_APP'
);
```

- 상품 등록, 주문 등록, 회원 정보 수정등 모든 데이터의 이력을 하나의 테이블에서 관리한다.
- `table_name`을 통해 구분할 수 있다.

데이터 조회

전체 감사 로그 조회

```
SELECT audit_id, table_name, record_id, action, changed_at, changed_by,
source_system
FROM audit_log
ORDER BY changed_at DESC;
```

[실행 결과]

audit_id	table_name	record_id	action	changed_at	changed_by	sourc
3	member	100	UPDATE	2026-03-15 15:00:00	member_100	MOBI
2	orders	1001	INSERT	2026-03-15 14:30:00	member_100	WEB_
1	product	1	INSERT	2026-03-15 10:00:00	admin_kim	WEB_

모든 테이블의 변경 이력을 한 곳에서 볼 수 있다.

특정 테이블의 이력 조회

```
-- 상품 테이블의 변경 이력
SELECT audit_id, record_id, action, new_data, changed_at, changed_by
FROM audit_log
WHERE table_name = 'product'
ORDER BY changed_at;
```

[실행 결과]

audit_id	record_id	action	new_data	changed_at	changed_by
1	1	INSERT	{"product_id": 1, "name": "스마트폰 케이스", "price": 15000...}	2026-03-15 10:00:00	admin_kim

특정 레코드의 이력 조회

```
-- 상품 1번의 변경 이력
SELECT audit_id, action, old_data, new_data, changed_at, changed_by
FROM audit_log
WHERE table_name = 'product' AND record_id = '1'
ORDER BY changed_at;
```

특정 사용자의 모든 활동 조회

```
-- admin_kim이 변경한 모든 내역
SELECT audit_id, table_name, record_id, action, changed_at
FROM audit_log
WHERE changed_by = 'admin_kim'
ORDER BY changed_at DESC;
```

[실행 결과]

audit_id	table_name	record_id	action	changed_at

1	product	1	INSERT	2026-03-15 10:00:00

장점

장점 1: 중앙 집중 관리

모든 테이블의 변경 이력을 한 곳에서 관리하고 조회할 수 있다.

장점 2: 구현이 단순하다

테이블마다 이력 테이블을 만들 필요가 없다. 하나의 테이블과 하나의 로직으로 처리한다.

장점 3: 전체 활동 추적

"어떤 사용자가 무엇을 변경했는지"를 시스템 전체에서 편리하게 추적할 수 있다.

```
-- 특정 시간대의 모든 변경 내역
SELECT * FROM audit_log
WHERE changed_at BETWEEN '2026-03-15 00:00:00' AND '2026-03-15 23:59:59'
ORDER BY changed_at;
```

단점

단점 1: 특정 테이블의 상세 이력 조회가 불편하다

JSON에서 특정 필드를 추출해야 하므로 쿼리가 복잡해진다.

```
-- 상품 가격 변경 이력을 조회하려면 JSON 파싱이 필요하다
SELECT
    audit_id,
    JSON_EXTRACT(old_data, '$.price') AS old_price,
    JSON_EXTRACT(new_data, '$.price') AS new_price,
    changed_at
FROM audit_log
WHERE table_name = 'product' AND record_id = '1'
```

```
ORDER BY changed_at;
```

- JSON을 사용하는 방법은 뒤에서 배운다.

단점 2: 시점 복원이 어렵다

특정 시점의 데이터를 복원하려면 JSON을 파싱하고 재구성해야 한다.

단점 3: 테이블이 매우 커진다

모든 테이블의 변경이 하나의 테이블에 쌓이므로 빠르게 커진다.

단점 4: 인덱스 효율이 떨어질 수 있다

`(table_name, record_id)` 조합으로 조회해야 하므로 인덱스 효율이 테이블별 이력 테이블보다 떨어질 수 있다.

공통 이력 테이블의 적절한 용도

공통 이력 테이블은 다음과 같은 용도로 사용하면 좋다.

1. **전체 시스템 감사**: 규정 준수, 보안 감사 목적
2. **문제 발생 시 원인 추적**: "누가, 언제, 무엇을" 변경했는지 확인
3. **백업 저장소**: 일단 저장해두고, 문제가 생기면 확인하는 용도
4. **디버깅**: 개발/테스트 환경에서 데이터 변경 추적

중요한 점은 공통 이력 테이블만으로 모든 이력 요구사항을 해결하려고 하면 안 된다는 것이다.
비즈니스적으로 중요한 데이터는 전용 이력 테이블을 별도로 만드는 것이 좋다.

혼합 전략

실무에서는 다음과 같이 혼합해서 사용하는 경우가 많다.

데이터 중요도	이력 관리 방식
매우 중요 (가격, 주문 등)	전용 이력 테이블 (전체 행 스냅샷)
중요 (회원 정보 등)	전용 이력 테이블 또는 공통 이력 테이블
일반 (설정, 로그 등)	공통 이력 테이블 또는 로그 파일
낮음 (임시 데이터 등)	이력 관리 안 함

```
-- 예: 가격은 전용 이력 테이블로
-- 나머지는 공통 이력 테이블로

-- 상품 전용 이력
CREATE TABLE product_history (...);

-- 나머지는 공통 감사 로그
CREATE TABLE audit_log (...);
```

실무 주의사항: 로그와 데이터베이스

마지막으로 실무에서 정말 중요한 조언을 하나 하겠다. 실제로 이 문제 때문에 실무에서 장애가 많이 발생한다.
모든 로그를 데이터베이스에 남겨서는 안 된다. 이 점을 반드시 기억해야 한다.

1. 데이터베이스는 귀중한 자원이다

데이터베이스(RDB)는 웹 애플리케이션 아키텍처에서 가장 비싸고, 확장이 어려우며, 부하가 집중되는 핵심 자원(Bottleneck)이다. CPU, 메모리, 디스크 I/O 모든 측면에서 그렇다.

그런데 여기에 단순히 "사용자가 페이지를 조회했다", "API 요청이 들어왔다", "디버깅용 정보" 같은 일반적인 시스템 로그(Log)까지 모두 `INSERT` 한다고 가정해 보자.

- **문제점**: 의미 없는 로그 데이터가 쌓여 디스크 용량이 순식간에 부족해진다.
- **장애 발생**: 로그를 기록하느라 DB의 리소스를 다 써버려서, 정작 중요한 '주문', '결제' 트랜잭션이 느려지거나 실행되지 않는 주객전도 상황이 발생한다. 로그 때문에 쇼핑몰이 멈추는 것이다.

2. 트래픽에 따른 저장소 선택

트래픽 규모에 따라 로그 관리 전략을 다르게 가져가야 한다.

트래픽이 아주 적은 경우 (초기 스타트업, 내부 어드민)

- 접속자가 하루에 몇 명 안 된다면, 편의성을 위해 데이터베이스에 로그를 남길 수 있다. SQL로 쉽게 조회할 수 있다는 장점이 있기 때문이다.

트래픽이 많은 경우 (대부분의 서비스)

- **절대 데이터베이스에 일반 로그를 남기면 안 된다.**
- 일반적인 애플리케이션 로그, 접근 로그 등은 **파일(File)** 로 남기는 것이 정석이다.
- 파일로 남겨진 로그는 `ELK Stack` (Elasticsearch, Logstash, Kibana) 같은 별도의 로그 수집 및 분석 시스템을 구축하여 관리해야 한다.

3. 비즈니스 이력 vs 시스템 로그
따라서 우리는 '데이터'와 '로그'를 명확히 구분해야 한다.

- **비즈니스 이력 (Audit Data)**: 누가 언제 주문 상태를 바꿨는지, 상품 가격을 누가 수정했는지 같은 정보. 이는 데이터의 성격을 가지며, 법적 분쟁이나 CS 처리를 위해 **데이터베이스**에 저장하는 것이 맞다. (앞서 배운 이력 테이블들이 여기에 해당한다.)
- **시스템 로그 (System Log)**: 디버깅 정보, 에러 스택 트레이스, 단순 접속 기록 등. 이는 휘발성이 강하거나 데이터로서의 가치가 상대적으로 낮다. 이는 **로그 파일**로 관리해야 한다.

결론: 데이터베이스는 쓰레기통이 아니다. 꼭 필요한 비즈니스 이력만 데이터베이스에 남기고, 나머지는 로그 파일로 관리해라. 이것이 대용량 트래픽을 견디는 시스템 설계의 기본이다.

Q&A: 공통 이력 테이블(audit_log)의 저장 위치

질문: "방금 '데이터베이스에 로그를 남기지 말라'고 했는데, 그럼 우리가 설계한 공통 이력 테이블(`audit_log`)은 데이터베이스에 만들어야 하는가, 아니면 로그 파일로 남겨야 하는가?"

답변: **우리가 설계한 `audit_log` 테이블은 데이터베이스에 저장하는 것이 맞다.** 우리가 설계한 테이블과 일반 로그의 차이점을 이해해야 한다.

1. **이것은 '데이터'다**: 우리가 설계한 `audit_log`는 단순한 접속 기록이 아니라, 비즈니스 데이터가 어떻게 변경되었는지(`old_data`, `new_data`)를 담고 있는 **중요한 자산**이다.
2. **즉시 조회가 필요하다**: 보통 이런 이력은 고객센터(CS)나 관리자 페이지에서 "이 회원이 언제 정보를 수정했지?"하고 실시간으로 검색할 수 있어야 한다. 로그 파일로 남기면 이런 실시간 조회가 매우 어렵다.
3. **트랜잭션 정합성**: 데이터 변경과 이력 저장이 동시에 성공하거나 동시에 실패해야 한다. 이는 데이터베이스의 트랜잭션 기능을 이용해야만 보장할 수 있다.

하지만 주의할 점이 있다.
공통 이력 테이블은 모든 테이블의 변경 사항이 모이는 곳이므로 데이터가 엄청나게 빨리 쌓인다. 따라서 실무에서는 **일정 기간(예: 3개월, 6개월)이 지난 데이터는 파일이나 별도의 아카이브 DB로 옮기고(Archiving), 운영 DB에서는 삭제하는 전략**을 반드시 함께 수립해야 한다. DB에 계속 쌓아두기만 하면 언젠가 DB가 터진다.

정리

데이터 변경 이력 설계 정리

지금까지 데이터 변경 이력을 관리하는 다양한 방법들을 알아보았다. 각 방식의 특징을 정리해보자.

방식별 비교

방식	과거 값 보관	시점 조회	구현 난이도	용량	실무 사용 빈도
변경 추적 컬럼만	X	X	매우 쉬움	최소	높음 (기본)
이전 값 컬럼	△ (1개만)	X	쉬움	최소	낮음
현재 테이블 + is_current	O	어려움	보통	많음	낮음
현재 테이블 + valid_from/to	O	쉬움	보통	많음	보통
전체 행 스냅샷 이력 테이블	O	어려움	보통	많음	**매우 높음**
전체 행 스냅샷 이력 테이블 + valid_from/to	O	쉬움	보통	많음	**매우 높음**
컬럼 단위 변경 로그	O	매우 어려움	높음	적음	낮음
공통 이력 테이블	O	어려움	보통	많음	보통 (보조용)

실무 권장 조합

실무에서 가장 많이 사용하는 조합은 다음과 같다.

기본 구성

1. **모든 테이블**: 기본 변경 추적 컬럼 (`created_at`, `created_by`, `updated_at`, `updated_by` 등등)
2. **중요 테이블**: 전용 이력 테이블 (전체 행 스냅샷)
3. **전체 시스템**: 공통 이력 테이블 (선택)

이력 테이블이 필요한 데이터

다음과 같은 데이터는 전용 이력 테이블을 만드는 것이 좋다.

- **가격 정보**: 상품 가격, 할인율 등
- **상태 정보**: 주문 상태, 회원 등급 등
- **잔액/수량**: 계좌 잔액, 포인트, 재고 수량 등
- **계약/약관**: 계약 조건, 이용약관 동의 내역 등
- **설정**: 사용자 설정, 시스템 설정 등
- **권한**: 사용자 권한, 역할 등

이력 테이블이 필요 없는 데이터

다음과 같은 데이터는 기본 변경 추적 컬럼만으로 충분한 경우가 많다.

- **로그 데이터**: 이미 로그 자체가 이력
- **임시 데이터**: 세션, 캐시 등
- **변경이 거의 없는 마스터 데이터**: 코드 테이블 등
- **대용량 콘텐츠**: 상품 설명, 게시글 본문 등 (용량 문제)

실무 체크리스트

새로운 테이블을 설계할 때 다음을 확인하자.

기본 변경 추적 컬럼을 추가했는가? 다음 컬럼은 대부분의 모든 테이블에 적용하는 것이 좋다.

- `created_at`, `created_by`, `updated_at`, `updated_by` 등등

다음 질문에 하나라도 "예"라면 전용 이력 테이블을 고려하자.

1. **이 데이터의 변경 이력이 비즈니스에 중요한가?**
    - 가격 변경, 상태 변경, 잔액 변경 등은 이력이 중요하다.
2. **과거 값을 조회해야 하는가?**
    - "어제 가격이 얼마였지?"라는 질문에 답해야 하는가?
3. **특정 시점의 데이터가 필요한가?**
    - "2026년 1월 기준 재고"를 조회해야 하는가?
4. **감사(Audit) 요구사항이 있는가?**
    - 금융, 의료 등 규제 산업에서는 필수

학습 내용 정리

컬럼에 이전 값 보관 방식

- 현재 테이블에 `previous_price`와 같은 별도 컬럼을 만들어 직전 값을 저장하는 방식이다.
- 구현이 매우 간단하고 조인이 필요 없어 조회가 빠르다.
- 변경이 2회 이상 발생하면 직전 값 외의 과거 이력은 소실된다.
- 추적할 컬럼이 늘어날수록 테이블 구조가 복잡해진다.
- 전체 이력이 필요 없거나 변경이 거의 없는 데이터에 적합하다.

현재 테이블로 이력 관리 - 시작

- 데이터를 수정할 때 `UPDATE` 대신 새로운 행을 `INSERT` 하여 이력을 남긴다.
- `is_current` 컬럼을 사용하여 현재 유효한 행을 구분한다.
- 모든 변경 이력이 행으로 남고 이전 값을 확인할 수 있다.
- 특정 시점(예: 3월 10일 기준)의 데이터를 조회하거나 통계를 내기 위해 복잡한 서브쿼리와 조인이 필요하며 성능 문제가 발생할 수 있다.
- 단일 테이블에 데이터가 과도하게 쌓이는 문제가 있다.

현재 테이블로 이력 관리 - 유효 기간

- `valid_from`과 `valid_to` 컬럼을 추가하여 데이터의 유효 기간을 명시한다.
- 현재 데이터의 `valid_to`는 `9999-12-31`로 설정하여 조회 편의성을 높인다.
- 범위 검색(Between)을 통해 특정 시점 조회 쿼리가 매우 단순해지고 인덱스 활용이 좋아진다.
- 데이터 변경 시 기존 행의 `valid_to`를 `UPDATE` 하고 새 행을 `INSERT` 해야 하므로 트랜잭션 관리가 필요하다.
- `is_current` 컬럼은 중복 데이터이나, 쿼리 직관성과 인덱스 효율(성능 최적화)을 위해 반정규화하여 함께 사용하는 것이 좋다.
- 여전히 단일 테이블이 비대해지는 문제는 해결되지 않는다.

전체 행 스냅샷 이력 테이블 - 시작

- 원본 테이블(`product`)과 이력 테이블(`product_history`)을 물리적으로 분리한다.
- 실무 조회 트래픽의 99%인 현재 데이터 조회의 성능을 최적화할 수 있다.
- 이력 테이블에는 원본의 `created_at`(데이터 생성일)과 별도로 `history_created_at`(이력 기록일)을 둬야 한다.
- 변경 발생 시 이력 테이블에 당시의 전체 행 상태를 복사(스냅샷)하여 저장한다.

전체 행 스냅샷 이력 테이블 - 주의점

- **최초 데이터 등록(INSERT) 시점에도 반드시 이력 테이블에 데이터를 저장해야 한다.**
- 변경(`UPDATE`) 시점에만 이력을 남기면, 데이터의 생성 시점 기록이 누락되어 타임라인이 끊긴다.
- INSERT 시점부터 저장해야 조회 시 `UNION` 없이 이력 테이블만으로 완벽한 과거 조회가 가능하다.

- 원본 테이블 데이터가 삭제되어도 이력 테이블을 통해 복구가 가능하다.
- 저장 공간을 아끼는 것보다 개발 복잡도를 낮추고 데이터 정합성을 지키는 것이 더 중요하다.

전체 행 스냅샷 이력 테이블 - 유효 기간

- 분리된 이력 테이블에도 `valid_from`, `valid_to`를 적용하여 시점 조회를 최적화할 수 있다.
- 마스터 데이터(상품, 회원 등 상태 유지 데이터)는 유효 기간 컬럼을 사용하는 것이 권장된다.
- 트랜잭션 데이터(주문, 로그 등 사건 기록 데이터)는 변경이 빈번하고 양이 많으므로 유효 기간 없이 `INSERT`만 하는 것이 유리하다.
- 유효 기간 컬럼 사용 시 데이터 변경마다 이전 이력의 `UPDATE`가 필요하므로 트랜잭션 비용이 발생함을 고려해야 한다.

전체 행 스냅샷 이력 테이블 - 한계

- 컬럼 하나만 바뀌어도 전체 행을 복사하므로 중복 데이터가 발생하고 용량을 많이 차지한다.
- 변경된 컬럼이 무엇인지 직관적으로 알기 어렵다.
- 그러나 저장 공간 비용보다 개발 및 유지보수 비용(인건비)이 훨씬 비싸므로, 특별한 이유가 없다면 **전체 행 스냅샷 방식을 기본(Default)으로 사용**하는 것이 정답이다.
- 복원이 쉽고, 쿼리가 단순하며, 다양한 분석이 가능하다.

컬럼 단위 변경 로그 테이블

- 변경된 컬럼만(`column_name`, `old_value`, `new_value`) 저장하여 용량을 최소화한다.
- 무엇이 변경되었는지 파악하기 쉽다.
- 특정 시점의 전체 데이터를 복원하려면 엄청나게 복잡한 쿼리와 비용이 든다.
- 타입 정보가 손실되며(문자열 저장), 통계 쿼리 작성이 어렵다.
- 데이터 용량이 감당 불가능할 정도로 크거나, 특정 필드의 변경 내역만 추적하면 되는 특수한 경우에만 사용한다.

공통 이력 테이블

- 모든 테이블의 변경 이력을 하나의 `audit_log` 테이블에서 통합 관리한다.
- `table_name`, `record_id`, `old_data(JSON)`, `new_data(JSON)` 등을 저장한다.
- 시스템 전체의 활동 추적(감사)이나 디버깅용으로 유용하다.
- JSON 파싱이 필요하여 상세 조회나 시점 복원이 어렵고, 테이블이 매우 빠르게 커진다.
- 중요 데이터(주문, 결제 등)는 전용 이력 테이블을 사용하고, 일반 데이터는 공통 이력 테이블을 사용하는 혼합 전략이 유효하다.

데이터 변경 이력 설계 정리

- **데이터베이스에는 비즈니스 데이터(이력)만 저장하고, 단순 시스템 로그(접속 기록, 디버깅 정보)는 파일로 저장해야 한다.**
- 데이터베이스에 모든 로그를 쌓으면 리소스 부족으로 정작 중요한 트랜잭션 처리에 장애가 발생할 수 있다.

- 파일 로그는 ELK 스택 등을 이용해 별도로 관리한다.
- 공통 이력 테이블(`audit_log`)은 비즈니스 자산이므로 DB에 저장하되, 주기적으로 아카이빙하고 운영 DB에서 삭제하여 용량을 관리해야 한다.
- **기본 전략은 전체 행 스냅샷(Insert 시점 포함)**이며, 상황에 따라 다른 방식을 고려한다.