10. JSON 설계

#0.강의/2.데이터베이스로드맵/4.설계2

- /EAV의 한계와 JSON의 필요성
- /JSON 문법
- /MySQL에서 JSON 사용하기 1
- /MySQL에서 JSON 사용하기 2
- /JSON 활용 - 다양한 실무 사례 1
- /JSON 활용 - 다양한 실무 사례 2
- /JSON 인덱스와 성능 최적화 1
- /JSON 인덱스와 성능 최적화 2
- /JSON 설계의 장단점과 한계
- /JSON 사용 가이드라인
- /관계형 데이터베이스 vs NoSQL
- /정리

EAV의 한계와 JSON의 필요성

이전 수업에서 EAV(Entity-Attribute-Value) 패턴을 학습했다. EAV는 유연한 속성 관리가 가능하다는 장점이 있지만, 실무에서 사용하다 보면 여러 가지 한계에 부딪히게 된다. 이번 수업에서는 EAV의 한계를 먼저 살펴보고, 그 대안으로 등장한 JSON 설계에 대해 알아보겠다.

EAV 패턴의 실무 한계

EAV 패턴을 실제 프로젝트에 적용했을 때 발생하는 문제점들을 구체적으로 살펴보자.

복잡한 쿼리

EAV에서 상품 정보를 한 줄에 조회하려면 여러 행을 하나의 행으로 피벗(Pivot)해야 한다. 이전에 만든 EAV 테이블을 다시 확인해보자.

```
-- EAV 테이블 생성
DROP TABLE IF EXISTS product_attribute;
DROP TABLE IF EXISTS product;
```

```sql
CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    created_at DATETIME NOT NULL
);

CREATE TABLE product_attribute (
    attribute_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    product_id BIGINT NOT NULL,
    attr_name VARCHAR(100) NOT NULL,
    attr_value VARCHAR(500) NOT NULL,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- 샘플 데이터 입력
INSERT INTO product (name, category, created_at) VALUES
('갤럭시 S24', '스마트폰', NOW()),
('맥북 프로 16', '노트북', NOW()),
('소니 WH-1000XM5', '헤드폰', NOW());

INSERT INTO product_attribute (product_id, attr_name, attr_value) VALUES
-- 갤럭시 S24 속성
(1, 'screen_size', '6.2'),
(1, 'storage', '256'),
(1, 'color', '팬텀 블랙'),
(1, 'battery', '4000'),
(1, 'ram', '8'),
-- 맥북 프로 16 속성
(2, 'screen_size', '16.2'),
(2, 'storage', '512'),
(2, 'color', '스페이스 그레이'),
(2, 'cpu', 'M5 Pro'),
(2, 'ram', '18'),
-- 소니 헤드폰 속성
(3, 'color', '블랙'),
(3, 'noise_canceling', 'true'),
(3, 'battery_life', '30'),
(3, 'driver_size', '30');
```

이제 스마트폰의 화면 크기와 저장 용량을 조회한다고 가정해보자. EAV에서는 다음과 같이 복잡한 쿼리가 필요하다.

```sql
-- EAV에서 특정 속성들을 열로 변환하여 조회
SELECT
    p.product_id,
    p.name,
    MAX(CASE WHEN pa.attr_name = 'screen_size' THEN pa.attr_value END) AS
screen_size,
    MAX(CASE WHEN pa.attr_name = 'storage' THEN pa.attr_value END) AS storage,
    MAX(CASE WHEN pa.attr_name = 'color' THEN pa.attr_value END) AS color,
    MAX(CASE WHEN pa.attr_name = 'ram' THEN pa.attr_value END) AS ram
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE p.category = '스마트폰'
GROUP BY p.product_id, p.name;
```

[실행 결과]

product_id	name	screen_size	storage	color	ram
1	갤럭시 S24	6.2	256	팬텀 블랙	8

보다시피 단순히 상품 정보를 조회하는 것인데도 `CASE WHEN`과 `GROUP BY`를 사용한 복잡한 피벗 쿼리가 필요하다. 속성이 10개, 20개로 늘어나면 쿼리는 더욱 길어진다.

데이터 타입 문제

EAV의 가장 큰 문제 중 하나는 모든 값이 문자열(`VARCHAR`)로 저장된다는 점이다. 숫자 비교가 필요한 상황을 보자.

```sql
-- 저장 용량이 300GB 이상인 상품 조회 (잘못된 결과)
SELECT p.name, pa.attr_value AS storage
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE pa.attr_name = 'storage'
  AND pa.attr_value >= '300';
```

[실행 결과]

name	storage

맥북 프로 16	512

이 쿼리는 언뜻 보면 정상적으로 동작하는 것 같지만, 문자열 비교이기 때문에 위험하다. 다음 예시를 보자.

```
-- 문자열 비교의 위험성 확인
INSERT INTO product (name, category, created_at) VALUES ('테스트폰', '스마트폰',
NOW());
INSERT INTO product_attribute (product_id, attr_name, attr_value) VALUES (4,
'storage', '64');

-- 다시 조회
SELECT p.name, pa.attr_value AS storage
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE pa.attr_name = 'storage'
  AND pa.attr_value >= '300';
```

[실행 결과]

name	storage
맥북 프로 16	512
테스트폰	64

64가 300보다 크다고? 문자열 비교에서는 '6'이 '3'보다 크기 때문에 이런 결과가 나온다. 이를 해결하려면 매번 형변환이 필요하다.

```
-- 올바른 숫자 비교를 위한 형변환
SELECT p.name, pa.attr_value AS storage
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
WHERE pa.attr_name = 'storage'
  AND CAST(pa.attr_value AS UNSIGNED) >= 300;
```

[실행 결과]

name	storage
맥북 프로 16	512

이렇게 형변환을 해야 올바른 결과가 나온다. 하지만 매번 `CAST`를 사용해야 하고, 인덱스도 제대로 활용되지 않는다. 물론 대안으로 별도의 숫자 타입 컬럼을 추가할 수 있지만, 하나의 로우(Row)에 사용하지 않는 나머지 타입 컬럼들은 `NULL`로 채워지게 된다는 점은 감수해야 한다.

성능 문제

EAV는 데이터가 늘어날수록 성능이 급격히 저하된다. 상품 1개에 속성이 10개라면, 상품 10만 개에 속성은 100만 행이 된다. 여기에 JOIN과 GROUP BY까지 수행하면 성능 문제가 심각해진다.

JSON이 필요한 이유

EAV의 이러한 한계를 해결하기 위해 현대 관계형 데이터베이스들은 JSON 타입을 지원하기 시작했다. JSON을 사용하면 다음과 같은 장점이 있다. 혹시 JSON을 잘 모른다면 바로 뒤에서 자세히 설명하니 걱정하지 않아도 된다.

단순한 데이터 구조

EAV는 속성 하나당 행 하나가 필요하지만, JSON은 하나의 컬럼에 모든 속성을 저장할 수 있다.

```
DROP TABLE IF EXISTS product_json;

-- JSON을 사용한 상품 테이블 (미리보기)
CREATE TABLE product_json (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    attributes JSON, -- 여기에 저장
    created_at DATETIME NOT NULL
);

INSERT INTO product_json (name, category, attributes, created_at) VALUES
('갤럭시 S24', '스마트폰',
 '{"screen_size": 6.2, "storage": 256, "color": "팬텀 블랙", "battery": 4000,
"ram": 8}',
```

```
    NOW());
```

EAV에서는 5개의 행이 필요했던 속성이 JSON에서는 하나의 컬럼 값으로 저장된다.

네이티브 데이터 타입

JSON에서는 숫자는 숫자로, 문자열은 문자열로, 불리언은 불리언으로 저장된다. 형변환 없이 올바른 비교가 가능하다.

간단한 쿼리

피벗 쿼리 없이 JSON 함수만으로 데이터를 쉽게 추출할 수 있다.

```
-- JSON에서 속성 조회 (미리보기)
SELECT
    name,
    attributes->>'$.screen_size' AS screen_size,
    attributes->>'$.storage' AS storage
FROM product_json
WHERE category = '스마트폰';
```

EAV의 복잡한 피벗 쿼리에 비해 훨씬 직관적이다.

관계형 데이터베이스와 JSON의 역사

관계형 데이터베이스에서 JSON 지원은 비교적 최근에 시작되었다. 그 배경을 이해하면 JSON 설계를 더 잘 활용할 수 있다.

NoSQL의 등장과 영향

2000년대 후반, MongoDB, CouchDB 같은 문서 지향(Document-Oriented) 데이터베이스들이 등장했다. 이들은 스키마 없이 유연하게 데이터를 저장할 수 있다는 점에서 많은 인기를 얻었다. 특히 다음과 같은 상황에서 강점을 보였다.

- 속성이 자주 변경되는 데이터
- 중첩된 복잡한 데이터 구조
- 빠른 개발과 프로토타이핑

관계형 데이터베이스 진영에서는 이러한 유연성을 제공하기 위해 JSON 타입을 도입하기 시작했다.
참고로 MySQL, PostgreSQL, Oracle, SQL Server와 같은 주요 데이터베이스들의 최신 버전은 대부분 JSON 타입을 지원한다.

SQL 표준과 JSON

JSON은 SQL:2016 표준에 공식적으로 포함되었다. `JSON_VALUE`, `JSON_QUERY`, `JSON_TABLE` 등의 함수가 표준으로 정의되었다. 다만 각 데이터베이스마다 구현 방식과 함수명에 차이가 있으므로, 사용하는 데이터베이스의 문서를 확인하는 것이 중요하다.

이번 수업 정리

이번 수업에서는 EAV의 한계와 JSON이 필요한 이유를 학습했다.

EAV의 주요 한계는 다음과 같다.

- 복잡한 피벗 쿼리 필요
- 모든 값이 문자열로 저장되어 타입 문제 발생
- 데이터 증가에 따른 성능 저하
- 참조 무결성 보장 불가

JSON 설계의 장점은 다음과 같다.

- 하나의 컬럼에 여러 속성 저장 가능
- 네이티브 데이터 타입 지원 (숫자, 문자열, 불리언 등)
- 간단한 JSON 함수로 데이터 추출
- 중첩된 구조 표현 가능

다음 수업에서는 JSON 문법의 기본을 학습하고, MySQL에서 JSON을 사용하는 구체적인 방법을 알아보겠다.

JSON 문법

JSON(JavaScript Object Notation)은 **데이터를 표현하는 경량 형식**이다. 원래 JavaScript에서 시작되었지만, 지금은 거의 모든 프로그래밍 언어와 데이터베이스에서 지원하는 표준 데이터 형식이 되었다. 데이터베이스에서 JSON을 제대로 활용하려면 먼저 JSON 문법을 이해해야 한다.

사실 JSON이 여러 곳에서 사용되는 또 하나의 이유는 정말 단순하기 때문이다.
이미 JSON에 대해서 아는 개발자 분들도 많이 있을 것이므로 여기서는 간단하게 설명하겠다.

JSON의 기본 구조

JSON은 두 가지 기본 구조로 이루어진다.

객체(Object)

중괄호 `{}`로 감싸며, 키-값 쌍으로 데이터를 표현한다. 키는 반드시 큰따옴표로 감싼 문자열이어야 한다.

```
{
  "name": "갤럭시 S24",
  "price": 1200000,
  "in_stock": true
}
```

배열(Array)

대괄호 `[]`로 감싸며, 순서가 있는 값들의 목록을 표현한다.

```
["빨강", "파랑", "초록"]
```

```
[1, 2, 3, 4, 5]
```

JSON 데이터 타입

JSON은 6가지 데이터 타입을 지원한다. EAV에서 모든 값이 문자열이었던 것과 달리, JSON은 다양한 타입을 네이티브로 지원한다.

문자열(String)

큰따옴표로 감싼다. 작은따옴표는 사용할 수 없다.

```
{
  "product_name": "맥북 프로",
```

```
  "description": "애플의 프로페셔널 노트북"
}
```

숫자(Number)

정수와 실수 모두 표현 가능하다. 따옴표 없이 작성한다.

```
{
  "price": 2500000,
  "weight": 1.83,
  "discount_rate": 0.15
}
```

불리언(Boolean)

`true` 또는 `false` 값을 가진다. 소문자로 작성하며, 따옴표를 사용하지 않는다.

```
{
  "is_available": true,
  "is_discontinued": false
}
```

null

값이 없음을 나타낸다. 소문자로 작성한다.

```
{
  "nickname": null,
  "secondary_phone": null
}
```

객체(Object)

중첩된 객체를 값으로 가질 수 있다.

```
{
  "product": {
    "name": "아이폰 15",
    "manufacturer": {
      "name": "Apple",
```

```
      "country": "미국"
    }
  }
}
```

배열(Array)

값으로 배열을 가질 수 있으며, 배열 안에는 어떤 타입이든 들어갈 수 있다.

```
{
  "colors": ["블랙", "화이트", "블루"],
  "specs": [
    {"name": "RAM", "value": 8},
    {"name": "Storage", "value": 256}
  ]
}
```

실무에서 자주 사용하는 JSON 패턴

실무에서 자주 접하게 되는 JSON 구조 패턴을 살펴보자.

상품 속성 표현

```
{
  "screen_size": 6.7,
  "storage": 256,
  "ram": 8,
  "color": "미드나잇",
  "5g_support": true,
  "release_date": "2026-03-15"
}
```

다국어 지원

```
{
  "name": {
    "ko": "무선 이어폰",
    "en": "Wireless Earbuds",
```

```
    "ja": "ワイヤレスイヤホン"
  }
}
```

옵션과 변형(Variants)

```
{
  "variants": [
    {"color": "블랙", "size": "M", "stock": 50, "sku": "BLK-M-001"},
    {"color": "블랙", "size": "L", "stock": 30, "sku": "BLK-L-001"},
    {"color": "화이트", "size": "M", "stock": 45, "sku": "WHT-M-001"}
  ]
}
```

메타데이터

```
{
  "seo": {
    "title": "갤럭시 S24 울트라 - 최저가 구매",
    "description": "삼성 갤럭시 S24 울트라를 최저가로 만나보세요",
    "keywords": ["갤럭시", "S24", "울트라", "삼성"]
  },
  "analytics": {
    "view_count": 15420,
    "last_viewed": "2026-01-15T15:30:00Z"
  }
}
```

JSON 작성 시 주의사항

JSON을 작성할 때 흔히 하는 실수들을 알아보자.

키는 반드시 큰따옴표 사용

```
// 잘못된 예시 (따옴표 없음)
{name: "상품명"}
```

```
// 잘못된 예시 (작은따옴표)
{'name': '상품명'}

// 올바른 예시
{"name": "상품명"}
```

마지막 요소 뒤에 쉼표 금지

```
// 잘못된 예시 (trailing comma)
{
  "name": "상품",
  "price": 10000,
}

// 올바른 예시
{
  "name": "상품",
  "price": 10000
}
```

주석 사용 불가

```
// 잘못된 예시 (주석 포함)
{
  "name": "상품",  // 상품명
  "price": 10000   /* 가격 */
}

// JSON은 주석을 지원하지 않는다
// 올바른 예시
{
  "name": "상품",
  "price": 10000
}
```

문자열 안의 특수문자

문자열 안에 큰따옴표나 역슬래시를 넣으려면 이스케이프(`\`) 처리가 필요하다.

```
{
  "message": "그가 말했다. \"안녕하세요\"",
  "path": "C:\\Users\\Documents"
}
```

MySQL에서 JSON 사용하기 1

이제 MySQL에서 JSON을 실제로 사용하는 방법을 알아보자. MySQL 5.7부터 JSON 타입을 지원하며, 8.0에서 많은 기능이 추가되었다.

JSON 컬럼이 있는 테이블 생성

먼저 JSON 컬럼을 가진 테이블을 생성해보자.

```
-- 기존 테이블 삭제
DROP TABLE IF EXISTS product_json;

-- JSON 컬럼이 있는 상품 테이블 생성
CREATE TABLE product_json (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    attributes JSON,
    created_at DATETIME NOT NULL DEFAULT NOW()
);
```

`attributes` 컬럼의 타입이 `JSON`인 것을 확인할 수 있다. MySQL은 이 컬럼에 유효한 JSON만 저장되도록 자동으로 검증한다.

JSON 데이터 삽입

JSON 데이터를 삽입하는 방법을 알아보자.

문자열로 직접 삽입

```
INSERT INTO product_json (name, category, attributes) VALUES
('갤럭시 S24', '스마트폰',
 '{"screen_size": 6.2, "storage": 256, "color": "팬텀 블랙", "ram": 8, "5g":
true}');
```

- JSON 데이터를 문자열로 직접 삽입하는 경우 JSON 형식을 자유롭게 입력할 수 있다.

JSON_OBJECT 함수 사용

`JSON_OBJECT` 함수를 사용하면 키-값 쌍을 직접 지정할 수 있다.

```
INSERT INTO product_json (name, category, attributes) VALUES
('아이폰 15 프로', '스마트폰',
 JSON_OBJECT(
   'screen_size', 6.1,
   'storage', 256,
   'color', '블랙 티타늄',
   'ram', 8,
   '5g', true
 ));
```

JSON_ARRAY 함수 사용

배열을 만들 때는 `JSON_ARRAY` 함수를 사용한다.

```
INSERT INTO product_json (name, category, attributes) VALUES
('맥북 프로 16', '노트북',
 JSON_OBJECT(
   'screen_size', 16.2,
   'storage', 512,
   'color', '스페이스 그레이',
   'ram', 18,
   'ports', JSON_ARRAY('HDMI', 'USB-C', 'MagSafe', 'SD카드')
 ));
```

데이터 확인

```
SELECT product_id, name, category, attributes FROM product_json;
```

[실행 결과]

product_id	name	category	attributes
1	갤럭시 S24	스마트폰	{"5g": true, "ram": 8, "color": "팬텀 블랙", "storage": 256, "screen_size": 6.2}
2	아이폰 15 프로	스마트폰	{"5g": true, "ram": 8, "color": "블랙 티타늄", "storage": 256, "screen_size": 6.1}
3	맥북 프로 16	노트북	{"ram": 18, "color": "스페이스 그레이", "ports": ["HDMI", "USB-C", "MagSafe", "SD카드"], "storage": 512, "screen_size": 16.2}

추가 샘플 데이터 입력

다양한 예제를 위해 더 많은 데이터를 입력하자.

```
INSERT INTO product_json (name, category, attributes) VALUES
('갤럭시 탭 S9', '태블릿',
 '{"screen_size": 11.0, "storage": 128, "color": "그라파이트", "ram": 8, "5g": false, "s_pen": true}'),

('소니 WH-1000XM5', '헤드폰',
 '{"color": "블랙", "noise_canceling": true, "battery_life": 30, "driver_size": 30, "bluetooth": "5.2"}'),

('LG 그램 17', '노트북',
 '{"screen_size": 17.0, "storage": 512, "color": "화이트", "ram": 16, "weight": 1.35, "ports": ["USB-C", "USB-A", "HDMI"]}'),

('에어팟 프로 2', '이어폰',
 '{"color": "화이트", "noise_canceling": true, "battery_life": 6, "case_battery": 30, "spatial_audio": true}'),
```

```
('갤럭시 워치 6', '스마트워치',
 '{"screen_size": 1.5, "color": "실버", "battery_life": 40, "water_resistant":
true, "gps": true}');
```

데이터 확인

```
SELECT product_id, name, category, attributes FROM product_json;
```

[실행 결과]

product_id	name	category	attributes
1	갤럭시 S24	스마트폰	{"5g": true, "ram": 8, "color": "팬텀 블랙", "storage": 256, "screen_size": 6.2}
2	아이폰 15 프로	스마트폰	{"5g": true, "ram": 8, "color": "블랙 티타늄", "storage": 256, "screen_size": 6.1}
3	맥북 프로 16	노트북	{"ram": 18, "color": "스페이스 그레이", "ports": ["HDMI", "USB-C", "MagSafe", "SD카드"], "storage": 512, "screen_size": 16.2}
4	갤럭시 탭 S9	태블릿	{"5g": false, "ram": 8, "color": "그라파이트", "s_pen": true, "storage": 128, "screen_size": 11.0}
5	소니 WH-1000XM5	헤드폰	{"color": "블랙", "bluetooth": "5.2", "driver_size": 30, "battery_life": 30, "noise_canceling": true}
6	LG 그램 17	노트북	{"ram": 16, "color": "화이트", "ports": ["USB-C", "USB-A", "HDMI"], "weight": 1.35, "storage": 512, "screen_size": 17.0}
7	에어팟 프로 2	이어폰	{"color": "화이트", "battery_life": 6, "case_battery": 30, "spatial_audio": true, "noise_canceling": true}

8	갤럭시 워치 6	스마트워치	{"gps": true, "color": "실버", "screen_size": 1.5, "battery_life": 40, "water_resistant": true}

JSON 데이터 조회

JSON 데이터를 조회하는 다양한 방법을 알아보자.

전체 JSON 조회

```sql
SELECT name, attributes FROM product_json WHERE product_id = 1;
```

[실행 결과]

name	attributes
갤럭시 S24	{"5g": true, "ram": 8, "color": "팬텀 블랙", "storage": 256, "screen_size": 6.2}

JSON_EXTRACT 함수로 특정 값 추출

`JSON_EXTRACT` 함수를 사용하면 JSON에서 특정 경로의 값을 추출할 수 있다. 경로는 `$`로 시작하며, `.`으로 키를 지정한다.

```sql
SELECT
    name,
    JSON_EXTRACT(attributes, '$.screen_size') AS screen_size,
    JSON_EXTRACT(attributes, '$.storage') AS storage,
    JSON_EXTRACT(attributes, '$.color') AS color
FROM product_json
WHERE category = '스마트폰';
```

[실행 결과]

name	screen_size	storage	color

갤럭시 S24	6.2	256	"팬텀 블랙"
아이폰 15 프로	6.1	256	"블랙 티타늄"

`JSON_EXTRACT`는 저장한 타입을 그대로 꺼낸다. **숫자는 숫자 타입으로, 불리언은 불리언 타입으로 조회**한다.
주의할 점은 `JSON_EXTRACT`로 문자열을 추출하면 큰따옴표가 포함된다는 것이다. `JSON_EXTRACT` 는 추출한 값의 **JSON 데이터 타입(형식)을 그대로 유지**하여 반환하기 때문이다. JSON 표준에서 문자열은 반드시 큰따옴표로 감싸져 있어야 하므로, 결과에도 큰따옴표가 포함된다.

> **JSON_QUERY, JSON_EXTRACT**
> MySQL에서는 표준 SQL과 달리 JSON_QUERY라는 함수명이 존재하지 않는다.
> 대신 **JSON_EXTRACT**가 그 역할을 수행한다.

➜ 연산자 (JSON_EXTRACT 축약형)
`->`는 `JSON_EXTRACT`의 축약형이다. 축약형이 간단하므로 주로 축약형을 사용한다.

```
SELECT
    name,
    attributes->'$.screen_size' AS screen_size,
    attributes->'$.storage' AS storage,
    attributes->'$.color' AS color
FROM product_json
WHERE category = '스마트폰';
```

[실행 결과]

name	screen_size	storage	color
갤럭시 S24	6.2	256	"팬텀 블랙"
아이폰 15 프로	6.1	256	"블랙 티타늄"

JSON_VALUE 함수로 특정 값 추출
JSON_VALUE 함수는 추출한 값에서 따옴표를 제거하고, **별도의 타입을 지정하지 않으면 문자열로 반환**한다.

```sql
SELECT
    name,
    JSON_VALUE(attributes, '$.screen_size') AS screen_size,
    JSON_VALUE(attributes, '$.storage') AS storage,
    JSON_VALUE(attributes, '$.color') AS color
FROM product_json
WHERE category = '스마트폰';
```

[실행 결과]

name	screen_size	storage	color
갤럭시 S24	6.2	256	팬텀 블랙
아이폰 15 프로	6.1	256	블랙 티타늄

- `screen_size`, `storage`, `color` 모두 문자열 타입으로 반환된다.

이제 `color` 값에 따옴표가 없다.

JSON_VALUE와 반환 타입 지정

`JSON_VALUE`의 강력한 기능 중 하나는 **반환되는 데이터의 타입을 우리가 원하는 대로 지정**할 수 있다는 점이다.
`JSON_VALUE` 함수에 별도의 타입을 지정하지 않으면 문자열로 반환한다.

숫자 연산을 하거나 크기를 비교할 때, 데이터 타입이 맞지 않으면 엉뚱한 결과가 나올 수 있다. 이때 `RETURNING` 절을 사용하여 데이터 타입을 명시적으로 변환해 준다.

이번에는 `storage`(저장 용량)를 정수형(`UNSIGNED`)으로, `screen_size`(화면 크기)를 실수형(`DECIMAL`)으로 변환해서 조회해보자.

```sql
SELECT
    name,
    JSON_VALUE(attributes, '$.storage') AS storage_text,
    JSON_VALUE(attributes, '$.storage' RETURNING UNSIGNED) AS storage,
    JSON_VALUE(attributes, '$.screen_size' RETURNING DECIMAL(4,2)) AS
screen_size
FROM product_json
WHERE category = '스마트폰'
ORDER BY storage DESC;
```

- `RETURNING`을 지정하지 않음: 문자로 반환한다. 여기서 반환한 `storage_text`의 256은 숫자가 아니라 문자열 256이다.
- `RETURNING UNSIGNED`: 추출한 값을 양의 정수(0과 양수)로 변환한다. 이제 `storage`는 문자열 256이 아니라 숫자 `256`이 된다.
- `RETURNING DECIMAL(3,2)` : 추출한 값을 소수점 둘째 자리까지 있는 숫자로 변환한다.

[실행 결과]

name	storage_text	storage	screen_size
갤럭시 S24	256	256	6.20
아이폰 15 프로	256	256	6.10

이렇게 타입을 지정하면 SQL 내부에서 대소 비교(`>`, `<`)를 하거나 정렬(`ORDER BY`)을 할 때, 문자열 기준이 아닌 숫자 기준으로 정확하게 동작하게 된다.

데이터 타입

`JSON_VALUE` 함수의 `RETURNING` 절 뒤에 사용할 수 있는 데이터 타입들은 **MySQL의 `CAST()` 함수에서 지원하는 타입**들과 거의 동일하다. 실무에서 자주 사용하는 타입들을 정리해두었으니 참고하자.

1. 숫자형 (Numeric Types)

가장 많이 사용되는 유형이다. JSON의 문자열 숫자를 실제 숫자로 변환한다.

타입	설명	사용 예시
`UNSIGNED [INTEGER]`	양수 정수 (0 포함)	`RETURNING UNSIGNED`
`SIGNED [INTEGER]`	음수 포함 정수	`RETURNING SIGNED`
`DECIMAL[(M[,D])]`	고정 소수점 (정확한 계산, 금액 등)	`RETURNING DECIMAL(10,2)`
`FLOAT`	부동 소수점 (단정밀도)	`RETURNING FLOAT`
`DOUBLE`	부동 소수점 (배정밀도)	`RETURNING DOUBLE`

2. 날짜 및 시간형 (Date and Time Types)

JSON에 저장된 날짜 문자열(`"2023-01-01"`)을 날짜 타입으로 인식시켜 **기간 검색** 등을 가능하게 한다.

타입	설명	사용 예시
`DATE`	날짜 (`YYYY-MM-DD`)	`RETURNING DATE`
`DATETIME`	날짜 + 시간	`RETURNING DATETIME`
`TIME`	시간 (`HH:MM:SS`)	`RETURNING TIME`

3. 문자열 및 바이너리 (String and Binary Types)

기본적으로 `JSON_VALUE`는 `VARCHAR(512)`를 반환하지만, 길이를 늘리거나 바이너리로 처리하고 싶을 때 사용한다.

타입	설명	사용 예시
`CHAR[(N)]`	고정 길이 문자열	`RETURNING CHAR(50)`
`BINARY[(N)]`	이진 문자열 (대소문자 구분 확실)	`RETURNING BINARY(16)`

->> 연산자 (따옴표 제거)

앞서 `JSON_VALUE`를 사용하면 따옴표가 제거된 값을 얻을 수 있다고 했다. 하지만 매번 `JSON_VALUE(col, '$.key')`라고 길게 쓰는 것은 번거롭다. 이를 위해 `->>` 연산자가 제공된다.

`->>`는 `JSON_VALUE`의 축약형이다. **추출한 값에서 따옴표를 제거하고 문자열로 반환**한다. 단, 이 연산자는 타입을 지정(`RETURNING`)할 수는 없으므로 단순히 문자 타입으로 값을 꺼낼 때 사용한다.

```
SELECT
    name,
    attributes->>'$.screen_size' AS screen_size,
    attributes->>'$.storage' AS storage,
    attributes->>'$.color' AS color
FROM product_json
WHERE category = '스마트폰';
```

[실행 결과]

name	screen_size	storage	color

갤럭시 S24	6.2	256	팬텀 블랙
아이폰 15 프로	6.1	256	블랙 티타늄

- `screen_size`, `storage`, `color` 모두 문자열 타입으로 반환된다.

결과를 보면 `color` 값에 따옴표가 제거된 것을 확인할 수 있다. 단순 조회가 목적이라면 `->>` 연산자를 가장 많이 사용하게 될 것이다. `->>` 는 문자열 타입을 반환하므로 숫자나 불리언을 비교해야 한다면 `->` 연산자를 사용해야 한다.

배열 요소 접근

JSON 데이터가 배열(Array) 형태라면 어떻게 접근해야 할까? 프로그래밍 언어와 마찬가지로 **인덱스(Index)**를 사용한다. 인덱스는 0부터 시작한다.

`product_json` 테이블의 노트북 데이터에는 `ports`라는 배열이 들어있다. 첫 번째 포트와 두 번째 포트 정보를 조회해보자.

노트북 데이터

product_id	name	category	attributes
3	맥북 프로 16	노트북	{"ram": 18, "color": "스페이스 그레이", "ports": ["HDMI", "USB-C", "MagSafe", "SD카드"], "storage": 512, "screen_size": 16.2}
6	LG 그램 17	노트북	{"ram": 16, "color": "화이트", "ports": ["USB-C", "USB-A", "HDMI"], "weight": 1.35, "storage": 512, "screen_size": 17.0}

```
SELECT
    name,
    attributes->>'$.ports[0]' AS first_port,
    attributes->>'$.ports[1]' AS second_port
FROM product_json
WHERE category = '노트북';
```

[실행 결과]

name	first_port	second_port
맥북 프로 16	HDMI	USB-C
LG 그램 17	USB-C	USB-A

`$.ports[0]`은 배열의 첫 번째 요소를, `$.ports[1]`은 두 번째 요소를 정확하게 가져온다. 이를 응용하면 특정 포트를 가진 노트북만 검색하는 것도 가능하다.

JSON 조건 검색

WHERE 절에서 JSON 값을 조건으로 사용하는 방법을 알아보자.

숫자 비교

```
-- 화면 크기가 6인치 이상인 제품 조회
SELECT name, attributes->'$.screen_size' AS screen_size
FROM product_json
WHERE attributes->'$.screen_size' >= 6;
```

[실행 결과]

name	screen_size
갤럭시 S24	6.2
아이폰 15 프로	6.1
맥북 프로 16	16.2
갤럭시 탭 S9	11.0
LG 그램 17	17.0

문자열 비교

```
-- 블랙 색상 제품 조회
SELECT name, attributes->>'$.color' AS color
```

```
FROM product_json
WHERE attributes->>'$.color' LIKE '%블랙%';
```

[실행 결과]

name	color
갤럭시 S24	팬텀 블랙
아이폰 15 프로	블랙 티타늄
소니 WH-1000XM5	블랙

불리언 비교

```
-- 노이즈 캔슬링 지원 제품 조회
SELECT name, category
FROM product_json
WHERE attributes->'$.noise_canceling' = true;
```

[실행 결과]

name	category
소니 WH-1000XM5	헤드폰
에어팟 프로 2	이어폰

JSON_CONTAINS로 배열 검색

배열에 특정 값이 포함되어 있는지 확인할 때는 `JSON_CONTAINS`를 사용한다.

```
-- USB-C 포트가 있는 노트북 조회
SELECT name, attributes->>'$.ports' AS ports
FROM product_json
WHERE JSON_CONTAINS(attributes->'$.ports', '"USB-C"');
```

[실행 결과]

name	ports
맥북 프로 16	["HDMI", "USB-C", "MagSafe", "SD카드"]
LG 그램 17	["USB-C", "USB-A", "HDMI"]

> `JSON_CONTAINS`의 두 번째 인자는 **JSON 형식이어야 하므로 문자열은** `'"USB-C"'` **처럼 큰따옴표를 포함해야 한다.**

JSON_CONTAINS_PATH로 키 존재 확인

특정 키가 존재하는지 확인할 때는 `JSON_CONTAINS_PATH`를 사용한다.

```
JSON_CONTAINS_PATH(json_doc, 'one' | 'all', path[, path] ...)
```

- **one** : 여러 path 중 하나라도 존재하면 TRUE
- **all** : 지정한 모든 path가 존재해야 TRUE

```sql
-- s_pen 속성이 있는 제품 조회
SELECT name, category, attributes->>'$.s_pen' AS s_pen
FROM product_json
WHERE JSON_CONTAINS_PATH(attributes, 'one', '$.s_pen');
```

[실행 결과]

name	category	s_pen
갤럭시 탭 S9	태블릿	true

MySQL에서 JSON 사용하기 2

JSON 데이터 수정

기존 JSON 데이터를 수정하는 방법을 알아보자.

JSON_SET으로 값 추가/수정

`JSON_SET`은 키가 있으면 값을 수정하고, 없으면 새로 추가한다.

```
JSON_SET(json_doc, path, value [, path, value] ...)
```

- **json_doc**: 대상 JSON 컬럼
- **path**: JSON 경로 ($.key, $.obj.key 등)
- **value**: 설정할 값 (문자, 숫자, boolean, JSON 등)

```
-- 갤럭시 S24에 새로운 속성 추가
UPDATE product_json
SET attributes = JSON_SET(attributes, '$.wireless_charging', true,
'$.storage', 1024)
WHERE product_id = 1;

-- 결과 확인
SELECT name, attributes FROM product_json WHERE product_id = 1;
```

- 기존에 없는 `wireless_charging` 속성과 값을 추가하고, 기존에 있는 `storage` 속성의 값을 256 ➜ 1024로 변경한다.

[실행 결과]

name	attributes
갤럭시 S24	{"5g": true, "ram": 8, "color": "팬텀 블랙", "storage": 1024, "screen_size": 6.2, "wireless_charging": true}

JSON_INSERT로 값 추가만

`JSON_INSERT`는 키가 없을 때만 추가한다. 이미 있는 키는 수정하지 않는다.

```sql
-- 이미 있는 color는 변경되지 않고, 없는 fast_charging만 추가됨
UPDATE product_json
SET attributes = JSON_INSERT(attributes, '$.color', '새로운 색상',
'$.fast_charging', true)
WHERE product_id = 1;

-- 결과 확인
SELECT name, attributes->>'$.color' AS color, attributes->>'$.fast_charging'
AS fast_charging
FROM product_json WHERE product_id = 1;
```

[실행 결과]

name	color	fast_charging
갤럭시 S24	팬텀 블랙	true

`color`는 기존 값 '팬텀 블랙'이 유지되고, `fast_charging`만 새로 추가되었다.

JSON_REPLACE로 값 수정만

`JSON_REPLACE`는 키가 있을 때만 수정한다. 없는 키는 추가하지 않는다.

```sql
-- 있는 storage는 변경되고, 없는 new_feature는 추가되지 않음
UPDATE product_json
SET attributes = JSON_REPLACE(attributes, '$.storage', 256, '$.new_feature',
true)
WHERE product_id = 1;

-- 결과 확인
SELECT name,
       attributes->>'$.storage' AS storage,
       attributes->>'$.new_feature' AS new_feature
FROM product_json WHERE product_id = 1;
```

[실행 결과]

name	storage	new_feature
갤럭시 S24	256	NULL

`storage`는 256로 변경되었지만, `new_feature`는 추가되지 않았다.

JSON_REMOVE로 값 삭제

```
-- fast_charging 속성 삭제
UPDATE product_json
SET attributes = JSON_REMOVE(attributes, '$.fast_charging',
'$.wireless_charging')
WHERE product_id = 1;

-- 결과 확인
SELECT name, attributes FROM product_json WHERE product_id = 1;
```

[실행 결과]

name	attributes
갤럭시 S24	{"5g": true, "ram": 8, "color": "팬텀 블랙", "storage": 256, "screen_size": 6.2}

- `fast_charging`, `wireless_charging`이 삭제되었다.

정리하면 다음과 같다.

함수	차이
JSON_SET	있으면 수정, 없으면 추가
JSON_INSERT	**없을 때만** 추가
JSON_REPLACE	**있을 때만** 수정
JSON_REMOVE	삭제

배열에 요소 추가

`JSON_ARRAY_APPEND`를 사용하면 배열에 요소를 추가할 수 있다.

```
-- 맥북에 새로운 포트 추가
UPDATE product_json
SET attributes = JSON_ARRAY_APPEND(attributes, '$.ports', '3.5mm 오디오')
WHERE product_id = 3;

-- 결과 확인
SELECT name, attributes->>'$.ports' AS ports FROM product_json WHERE
product_id = 3;
```

[실행 결과]

name	ports
맥북 프로 16	["HDMI", "USB-C", "MagSafe", "SD카드", "3.5mm 오디오"]

JSON 유틸리티 함수

자주 사용하는 JSON 유틸리티 함수들을 알아보자.

JSON_KEYS - 키 목록 조회

```
SELECT name, JSON_KEYS(attributes) AS attribute_keys
FROM product_json
WHERE product_id = 1;
```

[실행 결과]

name	attribute_keys
갤럭시 S24	["5g", "ram", "color", "storage", "screen_size"]

JSON_LENGTH - 요소 개수

```sql
-- 속성 개수 확인
SELECT name, JSON_LENGTH(attributes) AS attr_count
FROM product_json
LIMIT 5;
```

[실행 결과]

name	attr_count
갤럭시 S24	5
아이폰 15 프로	5
맥북 프로 16	5
갤럭시 탭 S9	6
소니 WH-1000XM5	5

```sql
-- 배열 길이 확인
SELECT name, JSON_LENGTH(attributes->'$.ports') AS port_count
FROM product_json
WHERE category = '노트북';
```

[실행 결과]

name	port_count
맥북 프로 16	5
LG 그램 17	3

JSON_TYPE - 타입 확인

```sql
SELECT
    name,
    JSON_TYPE(attributes->'$.screen_size') AS screen_type,
```

```
    JSON_TYPE(attributes->'$.color') AS color_type,
    JSON_TYPE(attributes->'$."5g"') AS `5g_type`
FROM product_json
WHERE product_id = 1;
```

- "5g"에 따옴표로 감싼 것을 주의하자.
- `5g_type` : 백틱이 추가되어 있다. MySQL에서 숫자로 시작하는 식별자는 백틱이 필요하다.

> ⚠ 주의
>
> $.key 형태의 key는 문자로 시작하고 문자/숫자/_만 허용한다.
>
> 숫자로 시작하는 키 또는 특수문자가 포함된 키는 반드시 따옴표로 감싸야 한다.
>
> JSON의 키는 가급적 영문 알파벳 문자로 시작하는 것이 좋다.

[실행 결과]

name	screen_type	color_type	5g_type
갤럭시 S24	DOUBLE	STRING	BOOLEAN

JSON_VALID - 유효성 검사

```
SELECT
    JSON_VALID('{"name": "test"}') AS valid_json,
    JSON_VALID('{name: test}') AS invalid_json,
    JSON_VALID('not json at all') AS not_json;
```

[실행 결과]

valid_json	invalid_json	not_json
1	0	0

JSON_PRETTY - 보기 좋게 출력

```
SELECT JSON_PRETTY(attributes) AS formatted_json
FROM product_json
```

```
WHERE product_id = 1;
```

[실행 결과]

formatted_json
{\n "5g": true,\n "ram": 8,\n "color": "팬텀 블랙",\n "price": 1199000,\n "storage": 256,\n "screen_size": 6.2\n}

- 실행 결과를 복사해서 붙여넣어 보면 `\n`(엔터)를 확인할 수 있다.

이번 수업 정리

이번 수업에서는 JSON 문법과 MySQL에서 JSON을 사용하는 기본 방법을 학습했다.

JSON 문법 핵심은 다음과 같다.

- 객체 `{}`와 배열 `[]` 두 가지 기본 구조
- 6가지 데이터 타입: 문자열, 숫자, 불리언, null, 객체, 배열
- 키는 반드시 큰따옴표로 감싸야 함

MySQL JSON 핵심 함수는 다음과 같다.

- `JSON_OBJECT`, `JSON_ARRAY`: JSON 생성
- `->`, `->>`: JSON 값 추출 (->는 JSON 타입, ->>는 문자열)
- `JSON_SET`, `JSON_INSERT`, `JSON_REPLACE`: JSON 수정
- `JSON_REMOVE`: JSON 값 삭제
- `JSON_CONTAINS`, `JSON_CONTAINS_PATH`: 포함 여부 확인

다음 수업에서는 JSON을 활용한 다양한 실무 사례와 고급 기능을 알아보겠다.

JSON 활용 - 다양한 실무 사례 1

이전 수업에서 JSON의 기본 문법과 MySQL에서 JSON을 다루는 기본 방법을 학습했다. 이번 수업에서는 실무에서 자주 접하는 다양한 JSON 활용 사례를 살펴보겠다. 실제 쇼핑몰 시스템을 예로 들어 JSON이 어떻게 활용되는지 구체적으로 알아보자.

사례 1: 카테고리별 다른 상품 속성

쇼핑몰에서 가장 흔히 접하는 문제다. 스마트폰, 노트북, 의류, 식품 등 카테고리마다 필요한 속성이 완전히 다르다. 이 문제를 JSON으로 어떻게 해결하는지 살펴보자.

테이블 설계

```
DROP TABLE IF EXISTS product_catalog;

CREATE TABLE product_catalog (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    base_price INT NOT NULL,
    attributes JSON,
    created_at DATETIME NOT NULL DEFAULT NOW()
);
```

공통 속성인 `name`, `category`, `base_price`는 일반 컬럼으로, 카테고리마다 다른 속성은 `attributes` JSON 컬럼으로 관리한다.

다양한 카테고리 상품 등록

```
-- 스마트폰
INSERT INTO product_catalog (name, category, base_price, attributes) VALUES
('갤럭시 S24 울트라', '스마트폰', 1650000, JSON_OBJECT(
    'screen_size', 6.8,
    'storage_options', JSON_ARRAY(256, 512, 1024),
    'colors', JSON_ARRAY('티타늄 그레이', '티타늄 블랙', '티타늄 바이올렛'),
    'ram', 12,
    '5g', true,
    'camera', JSON_OBJECT('main', 200, 'ultra_wide', 12, 'telephoto', 50),
    'battery', 5000,
    'os', 'Android 14'
));

-- 노트북
INSERT INTO product_catalog (name, category, base_price, attributes) VALUES
```

```
('맥북 프로 14 M3', '노트북', 2390000, JSON_OBJECT(
    'screen_size', 14.2,
    'storage_options', JSON_ARRAY(512, 1024, 2048),
    'colors', JSON_ARRAY('스페이스 그레이', '실버'),
    'ram_options', JSON_ARRAY(18, 36),
    'cpu', 'Apple M5 Pro',
    'gpu', 'Apple M5 Pro 14코어',
    'battery_life', 17,
    'weight', 1.55,
    'ports', JSON_ARRAY('HDMI', 'USB-C', 'MagSafe', 'SD카드')
));

-- 의류
INSERT INTO product_catalog (name, category, base_price, attributes) VALUES
('오버핏 맨투맨', '의류', 45000, JSON_OBJECT(
    'sizes', JSON_ARRAY('S', 'M', 'L', 'XL', 'XXL'),
    'colors', JSON_ARRAY('블랙', '화이트', '그레이', '네이비'),
    'material', '면 100%',
    'gender', 'unisex',
    'season', JSON_ARRAY('봄', '가을'),
    'care_instructions', JSON_ARRAY('세탁기 사용 가능', '표백제 사용 금지', '낮은 온도 다
림질')
));

-- 식품
INSERT INTO product_catalog (name, category, base_price, attributes) VALUES
('유기농 아보카도', '식품', 8900, JSON_OBJECT(
    'origin', '멕시코',
    'unit', '개',
    'quantity', 3,
    'organic', true,
    'storage', '냉장 보관',
    'shelf_life_days', 7,
    'nutrition', JSON_OBJECT(
        'calories', 160,
        'fat', 15,
        'protein', 2,
        'carbs', 9,
        'fiber', 7
    ),
    'allergens', JSON_ARRAY()
));
```

```
-- 가전제품
INSERT INTO product_catalog (name, category, base_price, attributes) VALUES
('LG 스탠바이미', '가전', 1290000, JSON_OBJECT(
    'screen_size', 27,
    'resolution', '1920x1080',
    'panel_type', 'IPS',
    'touch_screen', true,
    'battery_life', 3,
    'weight', 8.5,
    'speakers', JSON_OBJECT('type', '내장', 'output', '20W'),
    'connectivity', JSON_ARRAY('Wi-Fi', 'Bluetooth', 'HDMI', 'USB'),
    'os', 'webOS'
));
```

카테고리별 속성 조회

각 카테고리에 맞는 속성을 조회해보자.

```
-- 스마트폰 정보 조회
SELECT
    name,
    base_price,
    attributes->>'$.screen_size' AS screen_size,
    attributes->>'$.ram' AS ram,
    attributes->>'$.battery' AS battery,
    attributes->>'$.camera.main' AS main_camera
FROM product_catalog
WHERE category = '스마트폰';
```

[실행 결과]

name	base_price	screen_size	ram	battery	main_camera
갤럭시 S24 울트라	1650000	6.8	12	5000	200

```
-- 식품 영양정보 조회
SELECT
    name,
```

```
    attributes->>'$.origin' AS origin,
    attributes->>'$.nutrition.calories' AS calories,
    attributes->>'$.nutrition.protein' AS protein,
    attributes->>'$.organic' AS organic
FROM product_catalog
WHERE category = '식품';
```

[실행 결과]

name	origin	calories	protein	organic
유기농 아보카도	멕시코	160	2	true

이처럼 같은 테이블에서 카테고리마다 완전히 다른 속성을 관리할 수 있다.

사례 2: 상품 옵션과 재고 관리

쇼핑몰에서 하나의 상품이 여러 옵션(색상, 사이즈 조합)을 가지는 경우가 많다. 이를 JSON으로 관리하는 방법을 알아보자.

테이블 설계

```
DROP TABLE IF EXISTS product_variant;

CREATE TABLE product_variant (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    base_price INT NOT NULL,
    variants JSON NOT NULL,
    created_at DATETIME NOT NULL DEFAULT NOW()
);
```

옵션 데이터 등록

```
INSERT INTO product_variant (name, base_price, variants) VALUES
('나이키 에어맥스', 189000, JSON_ARRAY(
    JSON_OBJECT('sku', 'AM-BLK-250', 'color', '블랙', 'size', 250, 'stock', 15,
```

```
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-BLK-255', 'color', '블랙', 'size', 255, 'stock', 23,
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-BLK-260', 'color', '블랙', 'size', 260, 'stock', 18,
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-BLK-265', 'color', '블랙', 'size', 265, 'stock', 12,
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-WHT-250', 'color', '화이트', 'size', 250, 'stock', 8,
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-WHT-255', 'color', '화이트', 'size', 255, 'stock',
20, 'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-WHT-260', 'color', '화이트', 'size', 260, 'stock', 5,
'price_adjust', 0),
    JSON_OBJECT('sku', 'AM-LTD-260', 'color', '한정판 골드', 'size', 260,
'stock', 3, 'price_adjust', 50000)
));
```

저장 결과

product_id	name	base_price	variants
1	나이키 에어맥스	189000	[...]

여기서 variants에 저장된 JSON을 자세히 확인해보자.

product_id	variants
1	[{"sku": "AM-BLK-250", "size": 250, "color": "블랙", "stock": 15, "price_adjust": 0}, {"sku": "AM-BLK-255", "size": 255, "color": "블랙", "stock": 23, "price_adjust": 0}, {"sku": "AM-BLK-260", "size": 260, "color": "블랙", "stock": 18, "price_adjust": 0}, {"sku": "AM-BLK-265", "size": 265, "color": "블랙", "stock": 12, "price_adjust": 0}, {"sku": "AM-WHT-250", "size": 250, "color": "화이트", "stock": 8, "price_adjust": 0}, {"sku": "AM-WHT-255", "size": 255, "color": "화이트", "stock": 20, "price_adjust": 0}, {"sku": "AM-WHT-260", "size": 260, "color": "화이트", "stock": 5, "price_adjust": 0}, {"sku": "AM-LTD-260", "size": 260, "color": "한정판 골드", "stock": 3, "price_adjust": 50000}]

JSON 배열의 경우에는 다루기가 까다롭다. 이렇게 만들어진 JSON 배열을 마치 테이블처럼 편리하게 다루는 방법을 알아보자.

JSON_TABLE로 옵션 펼치기

MySQL 8.0의 `JSON_TABLE` 함수를 사용하면 JSON 배열을 별도의 테이블처럼 펼칠 수 있다.

JSON_TABLE 정의

```
JSON_TABLE(
  json_doc,
  path
  COLUMNS (
    column_name data_type PATH json_path [ERROR | EMPTY handling],
    ...
  )
)
```

`JSON_TABLE`은 JSON 데이터를 SQL 세계로 끌어오는 "가상 테이블 생성 함수"다.

- `json_doc`: JSON 컬럼 또는 JSON 값
- `path`: JSON_TABLE이 **반복할 기준 경로**, 보통 배열을 가리킨다.
  - 예)
    - `$[*]`: 루트에 있는 배열의 모든 요소
    - `$.orders[*]`: orders 배열의 모든 요소
- `COLUMNS`: JSON → 컬럼 매핑 정의
  - 예) `size INT PATH '$.size'` (`$.size`를 INT형 size 컬럼으로 매핑한다.)
- 오류 처리 옵션 (기본값: `NULL ON EMPTY`, `NULL ON ERROR`)

옵션	의미
ERROR ON ERROR	변환 실패 시 오류
NULL ON ERROR	실패 시 NULL
ERROR ON EMPTY	값 없음 → 오류
NULL ON EMPTY	값 없음 → NULL

예시를 통해서 JSON_TABLE의 사용법을 알아보자.

```sql
SELECT
    p.name,
    p.base_price,
    v.sku,
    v.color,
    v.size,
    v.stock,
    (p.base_price + v.price_adjust) AS final_price
FROM product_variant p,
JSON_TABLE(p.variants, '$[*]' COLUMNS (
    sku VARCHAR(50) PATH '$.sku',
    color VARCHAR(50) PATH '$.color',
    size INT PATH '$.size',
    stock INT PATH '$.stock',
    price_adjust INT PATH '$.price_adjust'
)) AS v;
```

이렇게 하면 `p`와 `v`는 조인을 통해서 결과를 생성한다.

[실행 결과]

name	base_price	sku	color	size	stock	final_price
나이키 에어맥스	189000	AM-BLK-250	블랙	250	15	189000
나이키 에어맥스	189000	AM-BLK-255	블랙	255	23	189000
나이키 에어맥스	189000	AM-BLK-260	블랙	260	18	189000
나이키 에어맥스	189000	AM-BLK-265	블랙	265	12	189000
나이키 에어맥스	189000	AM-WHT-250	화이트	250	8	189000
나이키 에어맥스	189000	AM-WHT-255	화이트	255	20	189000
나이키 에어맥스	189000	AM-WHT-260	화이트	260	5	189000
나이키 에어맥스	189000	AM-LTD-260	한정판 골드	260	3	239000

특정 조건의 옵션 찾기

```
-- 재고가 10개 이하인 옵션 찾기
SELECT
    p.name,
    v.sku,
    v.color,
    v.size,
    v.stock
FROM product_variant p,
JSON_TABLE(p.variants, '$[*]' COLUMNS (
    sku VARCHAR(50) PATH '$.sku',
    color VARCHAR(50) PATH '$.color',
    size INT PATH '$.size',
    stock INT PATH '$.stock'
)) AS v
WHERE v.stock <= 10;
```

[실행 결과]

name	sku	color	size	stock
나이키 에어맥스	AM-WHT-250	화이트	250	8
나이키 에어맥스	AM-WHT-260	화이트	260	5
나이키 에어맥스	AM-LTD-260	한정판 골드	260	3

색상별 총 재고 집계

```
SELECT
    p.name,
    v.color,
    SUM(v.stock) AS total_stock,
    COUNT(*) AS size_count
FROM product_variant p,
JSON_TABLE(p.variants, '$[*]' COLUMNS (
    color VARCHAR(50) PATH '$.color',
    stock INT PATH '$.stock'
)) AS v
GROUP BY p.name, v.color;
```

[실행 결과]

name	color	total_stock	size_count
나이키 에어맥스	블랙	68	4
나이키 에어맥스	화이트	33	3
나이키 에어맥스	한정판 골드	3	1

JSON 활용 - 다양한 실무 사례 2

사례 3: 사용자 설정 저장

사용자마다 다른 설정을 저장해야 할 때 JSON이 유용하다. 알림 설정, UI 설정, 개인화 설정 등이 대표적이다.

테이블 설계

```
DROP TABLE IF EXISTS user_settings;

CREATE TABLE user_settings (
    user_id BIGINT PRIMARY KEY,
    username VARCHAR(100) NOT NULL,
    settings JSON,
    created_at DATETIME NOT NULL DEFAULT NOW(),
    updated_at DATETIME NOT NULL DEFAULT NOW() ON UPDATE NOW()
);
```

사용자 설정 등록

```
INSERT INTO user_settings (user_id, username, settings) VALUES
(1, 'kim_user', JSON_OBJECT(
    'notifications', JSON_OBJECT(
        'email', true,
```

```
        'push', true,
        'sms', false,
        'marketing', false,
        'frequency', 'realtime'
    ),
    'display', JSON_OBJECT(
        'theme', 'dark',
        'language', 'ko',
        'timezone', 'Asia/Seoul',
        'items_per_page', 20
    ),
    'privacy', JSON_OBJECT(
        'profile_visible', true,
        'show_online_status', false,
        'allow_friend_requests', true
    )
)),
(2, 'lee_user', JSON_OBJECT(
    'notifications', JSON_OBJECT(
        'email', true,
        'push', false,
        'sms', true,
        'marketing', true,
        'frequency', 'daily'
    ),
    'display', JSON_OBJECT(
        'theme', 'light',
        'language', 'en',
        'timezone', 'America/New_York',
        'items_per_page', 50
    ),
    'privacy', JSON_OBJECT(
        'profile_visible', false,
        'show_online_status', false,
        'allow_friend_requests', false
    )
));
```

특정 설정 조회

```
SELECT
    username,
```

```
    settings->>'$.display.theme' AS theme,
    settings->>'$.display.language' AS language,
    settings->>'$.notifications.email' AS email_notification
FROM user_settings;
```

[실행 결과]

username	theme	language	email_notification
kim_user	dark	ko	true
lee_user	light	en	true

특정 설정만 업데이트

```
-- 테마만 변경
UPDATE user_settings
SET settings = JSON_SET(settings, '$.display.theme', 'auto')
WHERE user_id = 1;

-- 결과 확인
SELECT username, settings->>'$.display.theme' AS theme
FROM user_settings WHERE user_id = 1;
```

[실행 결과]

username	theme
kim_user	auto

사례 4: 로그와 이벤트 데이터

사용자 행동 로그, 시스템 이벤트 등 구조가 다양한 데이터를 저장할 때 JSON이 효과적이다.

테이블 설계

```
DROP TABLE IF EXISTS event_log;
```

```sql
CREATE TABLE event_log (
    log_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    event_type VARCHAR(100) NOT NULL,
    user_id BIGINT,
    event_data JSON NOT NULL,
    created_at DATETIME NOT NULL DEFAULT NOW(),
    INDEX idx_event_type (event_type),
    INDEX idx_user_id (user_id),
    INDEX idx_created_at (created_at)
);
```

다양한 이벤트 로그 등록

```sql
-- 페이지 조회 이벤트
INSERT INTO event_log (event_type, user_id, event_data) VALUES
('page_view', 1, JSON_OBJECT(
    'page', '/products/phone/galaxy-s24',
    'referrer', 'https://google.com',
    'duration_seconds', 45,
    'device', JSON_OBJECT('type', 'mobile', 'os', 'iOS', 'browser', 'Safari')
));

-- 검색 이벤트
INSERT INTO event_log (event_type, user_id, event_data) VALUES
('search', 1, JSON_OBJECT(
    'query', '갤럭시 케이스',
    'filters', JSON_OBJECT('category', '액세서리', 'price_max', 50000),
    'results_count', 42,
    'clicked_position', 3
));

-- 장바구니 추가 이벤트
INSERT INTO event_log (event_type, user_id, event_data) VALUES
('add_to_cart', 1, JSON_OBJECT(
    'product_id', 101,
    'product_name', '갤럭시 S24',
    'quantity', 1,
    'price', 1200000,
    'cart_total', 1200000
));
```

```sql
-- 구매 완료 이벤트
INSERT INTO event_log (event_type, user_id, event_data) VALUES
('purchase', 1, JSON_OBJECT(
    'order_id', 10001,
    'items_count', 2,
    'total_amount', 1270000,
    'payment_method', 'card',
    'coupon_used', 'WELCOME10'
));

-- 시스템 에러 이벤트
INSERT INTO event_log (event_type, user_id, event_data) VALUES
('error', NULL, JSON_OBJECT(
    'error_code', 'DB_ERROR',
    'message', 'Failed to connect to database',
    'server', 'db1',
    'stack_trace', '...',
    'level', 'critical'
));
```

이벤트 타입별 조회

```sql
-- 사용자 1의 검색 이벤트 분석
SELECT
    event_data->>'$.query' AS search_query,
    event_data->>'$.results_count' AS results,
    event_data->>'$.clicked_position' AS clicked_at,
    created_at
FROM event_log
WHERE event_type = 'search' AND user_id = 1;
```

[실행 결과]

search_query	results	clicked_at	created_at
갤럭시 케이스	42	3	2026-01-20 15:30:00

```sql
-- 사용자 1의 전체 이벤트 분석
```

```sql
SELECT
    event_type,
    event_data->>'$.page' AS page,
    event_data->>'$.query' AS search_query,
    event_data->>'$.results_count' AS results,
    event_data->>'$.clicked_position' AS clicked_at,
    event_data->>'$.product_name' AS name,
    event_data->>'$.total_amount' AS amount,
    created_at
FROM event_log
WHERE user_id = 1;
```

[실행 결과]

event_type	page	search_query	results	clicked_at	name	am
page_view	/products/phone/galaxy-s24	NULL	NULL	NULL	NULL	NU
search	NULL	갤럭시 케이스	42	3	NULL	NU
add_to_cart	NULL	NULL	NULL	NULL	갤럭시 S24	NU
purchase	NULL	NULL	NULL	NULL	NULL	127

```sql
-- 시스템 에러 조회
SELECT
    event_data->>'$.error_code' AS error_code,
    event_data->>'$.message' AS message,
    event_data->>'$.level' AS level,
    created_at
FROM event_log
WHERE event_type = 'error';
```

[실행 결과]

error_code	message	level	created_at
DB_ERROR	Failed to connect to database	critical	2026-01-20 15:30:00

이번 수업 정리

이번 수업에서는 실무에서 자주 사용하는 JSON 활용 사례들을 학습했다.

주요 활용 사례는 다음과 같다.

- 카테고리별 다른 상품 속성 관리
- 상품 옵션과 재고 관리 (JSON_TABLE 활용)
- 다국어 지원
- 사용자 설정 저장
- 주문 정보 스냅샷 보존
- 로그와 이벤트 데이터

핵심 기법은 다음과 같다.

- `JSON_TABLE`: JSON 배열을 테이블처럼 펼쳐서 조회
- `JSON_OBJECT`, `JSON_ARRAY`: 구조화된 JSON 생성
- `JSON_SET`, `JSON_INSERT`: 부분 업데이트
- 중첩 경로 접근: `$.parent.child.value`

다음 수업에서는 JSON 인덱스와 성능 최적화에 대해 알아보겠다.

JSON 인덱스와 성능 최적화 1

JSON은 스키마 없는 유연함을 제공하지만, 데이터가 쌓였을 때의 성능 문제는 반드시 고민해야 한다. 많은 학생들이 "JSON은 인덱스를 걸 수 없으니 느리지 않나요?"라고 질문한다. 결론부터 말하면 **JSON 내부의 데이터도 인덱스를 걸 수 있고, 매우 빠르게 조회할 수 있다.**

이번 수업에서는 MySQL에서 JSON 데이터의 검색 성능을 최적화하는 두 가지 핵심 방법인 **가상 컬럼(Virtual Column) 인덱스**와 **멀티 밸류(Multi-Valued) 인덱스**에 대해 학습한다.

JSON 검색 성능의 문제점

먼저 인덱스가 없는 상태에서 JSON 데이터를 검색할 때 어떤 일이 일어나는지 확인해보자.

우리가 만든 `product_json` 테이블에서 `storage`가 256인 상품을 찾는 상황을 가정해보자.

```
-- 실행 계획 확인 (인덱스 없음)
EXPLAIN
SELECT * FROM product_json
WHERE attributes->'$.storage' = 256;
```

[실행 결과]

id	select_type	table	type	possible_keys	key	rows	Extra
1	SIMPLE	product_json	**ALL**	NULL	NULL	8	Using

- `type: ALL` : 이것은 **풀 테이블 스캔(Full Table Scan)**을 의미한다. 데이터베이스가 테이블의 첫 번째 행부터 마지막 행까지 모든 JSON을 하나하나 열어서 `storage` 값을 확인했다는 뜻이다.
- 데이터가 8개일 때는 순식간이지만, 데이터가 100만 개라면 매우 느린 쿼리가 될 것이다.

일반적인 인덱스 생성 방식(`CREATE INDEX ... ON table(column)`)으로는 JSON 컬럼 전체를 인덱싱할 뿐, JSON 내부의 특정 키(`$.storage`)를 인덱싱할 수 없다. 이를 해결하기 위해 MySQL은 **가상 컬럼(Virtual Column)** 기능을 제공한다.

가상 컬럼(Virtual Column)을 활용한 인덱스

MySQL 5.7부터 지원하는 가상 컬럼을 사용하면 JSON 내부의 특정 필드를 마치 별도의 컬럼처럼 추출하고, 여기에 인덱스도 걸 수 있다.

1. 가상 컬럼 생성

먼저 JSON의 `storage` 값을 추출하는 가상 컬럼을 추가해보자.

```
ALTER TABLE product_json
ADD COLUMN v_storage INT GENERATED ALWAYS AS (attributes->'$.storage')
VIRTUAL;
```

- `GENERATED ALWAYS AS (...)` : 괄호 안의 식을 통해 값이 자동으로 생성된다는 뜻이다.
- `VIRTUAL` : 이 컬럼은 실제로 디스크에 저장되지 않고, 조회할 때만 계산된다. 따라서 추가적인 저장 공간을 거의 차지하지 않는다.

데이터를 조회해보면 `v_storage` 라는 컬럼이 생긴 것을 확인할 수 있다.

```
SELECT product_id, name, v_storage FROM product_json;
```

[실행 결과]

product_id	name	v_storage
1	갤럭시 S24	256
2	아이폰 15 프로	256
...	...	...

이 가상 컬럼을 사용하면 JSON의 `$.storage` 값을 일반적인 테이블의 컬럼처럼 접근할 수 있다.

왜 '가상(Virtual)' 컬럼인가?
여기서 `v_storage` 컬럼은 겉보기에는 일반적인 컬럼과 똑같아 보인다. 하지만 실제로는 테이블에 데이터가 저장되지 않는다. 이것이 **가상(Virtual)** 컬럼이라고 불리는 이유다.

일반적인 컬럼이라면 데이터를 추가할 때마다 디스크 공간을 차지하지만, 가상 컬럼은 **"이 컬럼의 값은 JSON의 `$.storage` 경로에서 가져온다"**라는 **계산식(메타데이터)**만 저장한다. 데이터베이스가 이 컬럼을 읽어야 할 때마다 원본 JSON을 참조해서 값을 그때그때 계산해서 보여주는 것이다. 마치 엑셀(Excel)에서 값이 고정된 셀이 아니라 수식이 걸려있는 셀과 비슷하다고 이해하면 된다.
여기서는 `(attributes->'$.storage')` 와 같은 수식이 걸려있는 것이다.

이 방식에는 두 가지 장점이 있다.

- **저장 공간 절약**: 데이터를 중복해서 저장하지 않으므로 디스크 공간을 아낄 수 있다. JSON 문서가 매우 크고 데이터가 많을수록 이 장점은 커진다.
- **데이터 무결성 유지**: 원본 JSON 데이터(`attributes`)가 변경되면 가상 컬럼(`v_storage`)의 값도 자동으로 변경된다. 개발자가 두 곳의 데이터를 맞추기 위해 별도로 업데이트 로직을 짤 필요가 없다.

이제 이 가상 컬럼을 활용해서 인덱스를 생성해보자. 가상 컬럼은 실제 데이터는 없지만, 일반 컬럼처럼 **인덱스(B-Tree)를 생성할 수 있다.** 이것이 JSON 최적화의 핵심이다.

2. 가상 컬럼에 인덱스 생성

이제 이 가상 컬럼에 일반적인 B-Tree 인덱스를 생성한다.

```
CREATE INDEX idx_v_storage ON product_json(v_storage);
```

3. 성능 개선 확인

이제 다시 조회를 해보자.

```
-- 가상 컬럼을 이용한 조회
EXPLAIN
SELECT * FROM product_json WHERE v_storage = 256;
```

[실행 결과]

id	select_type	type	key	rows	Extra
1	SIMPLE	**ref**	**idx_v_storage**	2	NULL

- `type: ref`: 풀 테이블 스캔이 사라지고 인덱스를 타게 되었다.
- `key: idx_v_storage`: 우리가 만든 인덱스가 사용되었다.

인덱스를 사용하는 것을 확인할 수 있을 것이다.

혹은 MySQL 옵티마이저가 똑똑하기 때문에, 원본 JSON 경로로 조회해도 식(Expression)이 일치하면 자동으로 인덱스를 사용한다.

```
```

```
-- JSON 경로로 조회해도 인덱스 적용됨
EXPLAIN
SELECT * FROM product_json
WHERE attributes->'$.storage' = 256;
```

이 방식을 사용하면 JSON의 유연함은 유지하면서도, 자주 조회하는 필드(예: `sku`, `price`, `category_id` 등)에 대해서는 관계형 데이터베이스의 강력한 성능을 그대로 누릴 수 있다.

VIRTUAL vs STORED

앞서 가상 컬럼을 생성할 때 `VIRTUAL`이라는 키워드를 사용했다. MySQL의 생성 컬럼(Generated Column)은 데이터를 저장하는 방식에 따라 **VIRTUAL**과 **STORED** 두 가지 타입으로 나뉜다.

VIRTUAL

```
ALTER TABLE product_json
ADD COLUMN v_storage INT GENERATED ALWAYS AS (attributes->'$.storage')
VIRTUAL;
```

- VIRTUAL은 기본값이다. 생략할 수 있다.

STORED

```
ALTER TABLE product_json
ADD COLUMN v_storage INT GENERATED ALWAYS AS (attributes->'$.storage') STORED;
```

실무에서는 대부분 `VIRTUAL`을 사용하며, `STORED`는 특별한 이유가 없다면 잘 사용하지 않는다. 여기서는 간단히 언급만 하겠다.

1. 차이점 비교

가장 큰 차이는 **"데이터를 디스크에 실제로 저장하느냐"**이다.

VIRTUAL (기본값)

- 데이터를 디스크에 저장하지 않는다. (메타데이터만 저장)
- `INSERT`나 `UPDATE` 시에는 계산하지 않고, 데이터를 `SELECT` 할 때마다 CPU가 계산해서 보여준다.
- 저장 공간을 차지하지 않아 효율적이다.

STORED

- 데이터를 디스크에 물리적으로 저장한다.
- `INSERT`나 `UPDATE` 될 때 미리 계산해서 저장해둔다.
- 저장 공간을 추가로 차지하지만, 읽을 때 CPU 연산이 필요 없다.

특징	VIRTUAL (가상)	STORED (저장)
저장 공간	거의 없음 (메타데이터만)	추가 공간 필요 (물리적 저장)
값 계산 시점	데이터를 읽을 때 (Read Time)	데이터를 쓸 때 (Write Time)
쓰기 성능	빠름 (계산 부하 적음)	느림 (계산 및 저장 부하 발생)
동기화	원본 변경 시 자동 반영	원본 변경 시 자동 업데이트
실무 활용도	**매우 높음**	낮음

2. 왜 실무에서는 VIRTUAL을 선호하는가?

"읽을 때마다 계산하면 조회가 느리지 않나요?"라는 의문이 들 수 있다. 하지만 여기에는 중요한 사실이 있다.

`VIRTUAL` **컬럼에 인덱스를 걸면, 그 인덱스 정보는 디스크에 물리적으로 저장된다.**

즉, 우리가 `idx_v_storage` 인덱스를 만드는 순간, 이미 `storage` 값들이 B-Tree 형태로 디스크에 예쁘게 정렬되어 저장된다는 뜻이다. 따라서 인덱스를 통해 조회할 때는 매번 계산하는 것이 아니라, 이미 저장된 인덱스 값을 바로 찾아가므로 `STORED`와 성능 차이가 거의 없다.

오히려 `STORED`를 사용하면 다음에서 2번 과정에 중복 낭비가 발생한다.

1. JSON 원본 데이터 저장
2. `STORED` 컬럼 데이터 저장 (중복 저장)
3. 인덱스 데이터 저장

굳이 2번 과정을 통해 디스크 용량을 낭비하고 쓰기 성능을 떨어뜨릴 이유가 없다. 따라서 JSON 데이터를 인덱싱할 때는 `VIRTUAL` **방식이 표준**이라고 생각하면 된다.

3. STORED는 언제 쓰는가?

`STORED` 방식은 JSON 추출 비용보다 훨씬 더 복잡하고 무거운 계산이 필요하거나, 해당 컬럼을 PK(Primary Key)로 설정해야 하는 등 제약 사항이 있을 때만 제한적으로 사용한다. 일반적인 JSON 필드 인덱싱 목적이라면 `VIRTUAL`만으로 충분하다.

JSON 인덱스와 성능 최적화 2

함수 기반 인덱스 (Function-Based Index)

앞서 배운 '가상 컬럼' 방식은 매우 유용하지만, 인덱스 하나를 만들기 위해 컬럼을 추가(`ALTER TABLE`)하고 다시 인덱스를 생성(`CREATE INDEX`)해야 하는 과정이 다소 번거롭게 느껴질 수 있다. "굳이 컬럼까지 만들어야 하나? 그냥 바로 인덱스만 걸면 안 되나?"라는 의문이 생길 수 있다.

MySQL 8.0.13 버전부터는 가상 컬럼을 명시적으로 만들지 않고도, **함수(수식)의 결과에 대해 직접 인덱스를 생성**할 수 있는 **함수 기반 인덱스** 기능을 지원한다.

1. 함수 기반 인덱스 생성

문법은 간단하다. 인덱스를 생성할 때 컬럼명 대신 `(수식)`을 이중 괄호로 감싸서 넣어주면 된다. `storage`에 대한 인덱스를 함수 기반으로 만들어보자.

```
-- 가상 컬럼 생성 과정 없이 바로 인덱스 생성
CREATE INDEX idx_func_storage
ON product_json (( CAST(attributes->'$.storage' AS UNSIGNED) ));
```

또는 다음과 같이 JSON_VALUE를 사용하면서 타입을 직접 지정할 수 있다.

```
-- 가상 컬럼 생성 과정 없이 바로 인덱스 생성 (CAST 없음)
CREATE INDEX idx_func_storage
ON product_json ((JSON_VALUE(attributes, '$.storage' RETURNING UNSIGNED)));
```

- `(( ... ))`: 수식을 인덱스로 사용할 때는 반드시 **이중 괄호**를 사용해야 한다.
- 타입을 지정해야 한다. (인덱스는 사용할 타입을 알아야 한다.)
- 내부적으로는 MySQL이 우리가 앞서 배운 '가상 컬럼'을 자동으로 생성하고 거기에 인덱스를 거는 방식으로 동작

한다. 즉, 성능상 차이는 없고 사용 편의성을 높인 기능(Syntactic Sugar)이다.

2. 실행 결과 확인

```
EXPLAIN
SELECT * FROM product_json
WHERE attributes->'$.storage' = 256;
```

[실행 결과]

id	select_type	table	type	key	rows	Extra
1	SIMPLE	product_json	**ref**	**idx_func_storage**	2	NULL

인덱스가 정상적으로 적용되는 것을 볼 수 있다.

실무에서는 가상 컬럼이 다른 조회 쿼리에서도 `SELECT` 절에 자주 등장한다면 **가상 컬럼** 방식을, 오직 `WHERE` 절의 검색 조건으로만 쓰인다면 **함수 기반 인덱스**를 사용하여 스키마를 깔끔하게 유지하는 것을 추천한다.

멀티 밸류 인덱스 (Multi-Valued Index)

가상 컬럼 인덱스는 값이 하나인 경우(1:1)에는 완벽하다. 하지만 JSON 배열 안에 있는 값을 검색해야 한다면 어떻게 해야 할까?

예를 들어, "HDMI 포트가 있는 노트북"을 찾고 싶다. `ports`는 `["HDMI", "USB-C"]`와 같은 배열이다. 기존 B-Tree 인덱스는 하나의 행에 하나의 인덱스 키만 가질 수 있어서 배열을 인덱싱할 수 없었다.

MySQL 8.0.17부터 도입된 **멀티 밸류 인덱스**는 하나의 행(JSON 배열)에 여러 개의 인덱스 키를 매핑할 수 있게 해준다.

1. 멀티 밸류 인덱스 생성

문법이 조금 독특하니 주의 깊게 봐야 한다. 앞서 배운 함수 기반 인덱스에 `CAST(... AS ... ARRAY)`를 사용해야 한다.

```
-- ports 배열에 대한 멀티 밸류 인덱스 생성
```

```
CREATE INDEX idx_ports
ON product_json (( CAST(attributes->'$.ports' AS CHAR(20) ARRAY) ));
```

- `CAST(... AS ... ARRAY)` : JSON 배열의 요소들을 지정한 타입(여기서는 문자, `CHAR(20)`)으로 변환하여 인덱싱하겠다는 의미다.
- 이중 괄호 `(( ... ))`를 사용하는 것에 주의하자.

2. 멀티 밸류 인덱스 활용 함수

멀티 밸류 인덱스를 타게 하려면 다음 함수들을 사용해야 한다.

- `MEMBER OF()` : 특정 값이 배열에 있는지 확인
- `JSON_CONTAINS()` : 특정 JSON이 포함되어 있는지 확인
- `JSON_OVERLAPS()` : 두 배열 간에 겹치는 요소가 있는지 확인

3. 성능 개선 확인

```
-- HDMI 포트가 있는 제품 검색
EXPLAIN
SELECT product_id, name, attributes->>'$.ports'
FROM product_json
WHERE 'HDMI' MEMBER OF(attributes->'$.ports');
```

[실행 결과]

id	select_type	table	type	key	rows	Extra
1	SIMPLE	product_json	**ref**	**idx_ports**	2	Using where

- `key: idx_ports` : 멀티 밸류 인덱스가 정상적으로 작동하여, 배열 내부의 값을 고속으로 검색했다.

`JSON_CONTAINS`를 사용할 수도 있다.

```
EXPLAIN
SELECT product_id, name
FROM product_json
WHERE JSON_CONTAINS(attributes->'$.ports', '"USB-C"');
```

- `JSON_CONTAINS`는 타겟 값이 JSON 형식이어야 하므로 문자열은 `'"USB-C"'`처럼 따옴표를 두 번 써야 한다.

JSON 인덱스 정리

이번 수업에서는 JSON 인덱스와 성능 최적화 기법을 학습했다.

JSON 인덱스 핵심은 다음과 같다.

- 가상 컬럼(Generated Column) + 인덱스로 JSON 값 검색 최적화
- 함수 기반 인덱스로 가상 컬럼 없이 인덱스 생성 가능 (MySQL 8.0.13+)
- Multi-Valued Index로 JSON 배열 검색 최적화 (MySQL 8.0.17+)

성능 최적화 전략은 다음과 같다.

- 자주 검색하는 속성은 가상 컬럼으로 추출하여 인덱스 적용
- 하이브리드 설계: 검색용 속성은 일반 컬럼, 유동적 속성은 JSON
- JSON 크기를 적절히 유지
- 전체 교체 대신 부분 업데이트 활용

인덱스 없는 JSON 검색은 전체 테이블 스캔을 유발하므로, 검색이 필요한 JSON 속성에는 반드시 인덱스 전략을 수립해야 한다.

다음 수업에서는 JSON 설계의 장단점과 한계, 그리고 실무에서의 적용 가이드라인을 알아보겠다.

JSON 설계의 장단점과 한계

지금까지 JSON의 기본 사용법, 다양한 활용 사례, 그리고 성능 최적화 방법을 학습했다. 이번 수업에서는 JSON 설계의 장단점과 한계를 명확히 이해하고, 실무에서 JSON을 언제 어떻게 사용해야 하는지 가이드라인을 제시하겠다.

JSON 설계의 장점

JSON 설계가 가져다주는 이점들을 구체적으로 살펴보자.

스키마 유연성

가장 큰 장점은 스키마 변경 없이 새로운 속성을 추가할 수 있다는 점이다.

```
DROP TABLE IF EXISTS product_flex;

CREATE TABLE product_flex (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    attributes JSON,
    created_at DATETIME NOT NULL DEFAULT NOW()
);

-- 초기 상품 등록
INSERT INTO product_flex (name, category, attributes) VALUES
('갤럭시 S24', '스마트폰', '{"brand": "삼성", "price": 1200000}');

-- 나중에 새로운 속성 추가 - ALTER TABLE 불필요
INSERT INTO product_flex (name, category, attributes) VALUES
('갤럭시 S25', '스마트폰', '{"brand": "삼성", "price": 1300000, "ai_features": ["통
역", "요약", "검색"], "satellite_sos": true}');

-- 두 상품 모두 정상 조회
SELECT name, attributes->>'$.brand' AS brand, attributes->>'$.ai_features' AS
ai_features
FROM product_flex;
```

[실행 결과]

name	brand	ai_features
갤럭시 S24	삼성	NULL
갤럭시 S25	삼성	["통역", "요약", "검색"]

일반 컬럼이었다면 `ALTER TABLE ADD COLUMN`이 필요했을 것이다. 대용량 테이블에서 `ALTER TABLE`은 상당한 시간이 소요될 수 있다.

복잡한 중첩 구조 표현

관계형 모델에서는 여러 테이블로 분리해야 하는 중첩 구조를 하나의 컬럼에 표현할 수 있다.

```
-- 일반 관계형 모델: 3개 테이블 필요
-- product, product_spec, product_spec_detail

-- JSON: 하나의 컬럼에 모두 표현
INSERT INTO product_flex (name, category, attributes) VALUES
('맥북 프로 16', '노트북', JSON_OBJECT(
    'brand', '애플',
    'price', 3500000,
    'specs', JSON_OBJECT(
        'display', JSON_OBJECT(
            'size', 16.2,
            'resolution', '3456x2234',
            'technology', 'Liquid Retina XDR',
            'refresh_rate', 120
        ),
        'processor', JSON_OBJECT(
            'name', 'M3 Max',
            'cores', JSON_OBJECT('performance', 12, 'efficiency', 4),
            'gpu_cores', 40
        ),
        'memory', JSON_OBJECT(
            'size', 36,
            'type', 'Unified Memory',
            'bandwidth', '400GB/s'
        )
    )
));

-- 중첩된 값 조회
SELECT
    name,
    attributes->>'$.specs.processor.name' AS cpu,
    attributes->>'$.specs.processor.cores.performance' AS p_cores,
    attributes->>'$.specs.memory.size' AS ram
FROM product_flex
WHERE category = '노트북';
```

[실행 결과]

name	cpu	p_cores	ram
맥북 프로 16	M3 Max	12	36

개발 생산성 향상

애플리케이션에서 객체나 JSON을 그대로 저장하고 조회할 수 있어 개발이 단순해진다.

```
-- 애플리케이션의 설정 객체를 그대로 저장
INSERT INTO product_flex (name, category, attributes) VALUES
('사용자 대시보드', '설정', '{
    "layout": {
        "columns": 3,
        "widgets": [
            {"id": "sales", "position": 1, "size": "large"},
            {"id": "visitors", "position": 2, "size": "medium"},
            {"id": "orders", "position": 3, "size": "medium"}
        ]
    },
    "theme": "dark",
    "refresh_interval": 30
}');
```

ORM이나 애플리케이션 코드에서 복잡한 매핑 없이 JSON 객체를 그대로 사용할 수 있다.

EAV 대비 장점

이전에 학습한 EAV와 비교했을 때 JSON의 장점을 정리해보자.

EAV vs JSON 비교

상품 하나에 속성 5개인 경우

- EAV: 6개 행 (상품 1행 + 속성 5행), JOIN 필요
- JSON: 1개 행, JOIN 불필요

EAV 사용

```
-- EAV
SELECT p.name,
```

```
       MAX(CASE WHEN pa.attr_name = 'brand' THEN pa.attr_value END) AS brand,
       MAX(CASE WHEN pa.attr_name = 'price' THEN pa.attr_value END) AS price
FROM product p
JOIN product_attribute pa ON p.product_id = pa.product_id
GROUP BY p.product_id, p.name;
```

JSON 사용

```
-- JSON
SELECT name,
       attributes->>'$.brand' AS brand,
       attributes->>'$.price' AS price
FROM product_flex;
```

JSON이 훨씬 직관적이고 간단하다.

JSON 설계의 단점

JSON의 단점과 한계도 명확히 이해해야 한다.

데이터 무결성 부재

JSON은 스키마가 없으므로 데이터베이스 수준에서 데이터 무결성을 보장하지 않는다.

```
-- 잘못된 데이터도 저장됨
INSERT INTO product_flex (name, category, attributes) VALUES
('잘못된 상품1', '스마트폰', '{"brand": "삼성", "price": "백만원"}'),  -- 가격이 문자열
('잘못된 상품2', '스마트폰', '{"brand": "삼성", "pricee": 1000000}'),-- 속성 오타
('잘못된 상품3', '스마트폰', '{"brand": "삼성"}');                    -- 필수 속성 누락

-- 모두 정상 저장됨
SELECT name, attributes FROM product_flex WHERE name LIKE '잘못된%';
```

[실행 결과]

name	attributes

잘못된 상품1	{"brand": "삼성", "price": "백만원"}
잘못된 상품2	{"brand": "삼성", "pricee": 1000000}
잘못된 상품3	{"brand": "삼성"}

일반 컬럼이었다면 `price INT NOT NULL`로 이런 문제를 방지할 수 있었다.

CHECK 제약조건으로 일부 보완

MySQL 8.0.16부터 JSON에 CHECK 제약조건을 사용할 수 있다.

```
DROP TABLE IF EXISTS product_validated;

CREATE TABLE product_validated (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    attributes JSON NOT NULL,
    -- JSON 유효성 검사
    CONSTRAINT chk_brand CHECK (JSON_CONTAINS_PATH(attributes, 'one',
'$.brand')),
    CONSTRAINT chk_price CHECK (
        JSON_CONTAINS_PATH(attributes, 'one', '$.price')
        AND JSON_TYPE(attributes->'$.price') = 'INTEGER'
    )
);

-- 정상 데이터
INSERT INTO product_validated (name, attributes) VALUES
('정상 상품', '{"brand": "삼성", "price": 1200000}');
```

[실행 결과]

```
Query OK, 1 row affected
```

```
-- brand 누락 - 실패
INSERT INTO product_validated (name, attributes) VALUES
```

```
('실패 상품', '{"price": 1200000}');
```

[실행 결과]

```
Error Code: 3819. Check constraint 'chk_brand' is violated.
```

```
-- price가 문자열 - 실패
INSERT INTO product_validated (name, attributes) VALUES
('실패 상품', '{"brand": "삼성", "price": "백만원"}');
```

[실행 결과]

```
Error Code: 3819. Check constraint 'chk_price' is violated.
```

하지만 CHECK 제약조건은 복잡한 검증에 한계가 있고, 성능 오버헤드도 있다.

참조 무결성 불가

JSON 내부의 값으로는 외래 키를 설정할 수 없다.

```
-- 일반 컬럼은 외래 키 가능
CREATE TABLE order_item (
    item_id BIGINT PRIMARY KEY,
    product_id BIGINT,
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- JSON 내부 값은 외래 키 불가능
-- {"product_id": 1} 이 값이 실제로 존재하는 product인지 DB가 검증 불가
```

집계와 분석의 어려움

JSON 데이터에 대한 집계 쿼리는 복잡하고 성능이 떨어진다.

```sql
-- 일반 컬럼: 단순 집계
SELECT category, AVG(price), COUNT(*)
FROM product
GROUP BY category;

-- JSON: 복잡한 집계
SELECT
    category,
    AVG(attributes->'$.price') AS avg_price,
    COUNT(*) AS cnt
FROM product_flex
WHERE attributes->'$.price' IS NOT NULL
GROUP BY category;
```

대규모 분석 작업에서는 JSON보다 일반 컬럼이 효율적이다.

왜 JSON 분석이 더 비효율적일까?
겉으로 보기에는 SQL 한 줄 차이처럼 보이지만, 데이터베이스 내부에서는 **비용(Cost)** 차이가 크다.

1. **파싱(Parsing) 오버헤드**: 일반 컬럼(`price`)은 데이터베이스가 해당 위치에 '숫자'가 있다는 것을 이미 알고 있어서 바로 읽어낼 수 있다. 하지만 JSON은 **매 행(Row)마다** JSON 문서를 열고, 내부 구조를 순회하여 `$.price`라는 키를 찾는 과정이 필요하다. 데이터가 100만 건이면 이 '찾는 과정'도 100만 번 반복된다. 비유하자면, **일반 컬럼은 진열장에 놓인 물건을 바로 집는 것이고, JSON은 포장된 택배 박스를 일일이 뜯어서 물건을 확인하는 것**과 같다. 이런 오버헤드 때문에 일반 컬럼보다 JSON이 약간은 더 느리다.
2. **데이터 용량과 I/O**: JSON은 값뿐만 아니라 키(`key`) 정보도 함께 저장한다. `{ "price": 10000 }`이라는 데이터는 단순히 숫자 `10000`만 저장하는 것보다 훨씬 많은 저장 공간을 차지한다. 분석을 위해 대량의 데이터를 메모리로 로드할 때, 불필요한 키 정보까지 함께 읽어야 하므로 디스크 I/O와 메모리 효율이 떨어진다.

결국 **대규모 데이터 분석이나 통계가 주 목적이라면, 해당 필드는 JSON 밖으로 꺼내어 정규화된 일반 컬럼으로 만드는 것이 정석**이다. JSON의 유연함은 '저장'과 '조회'에는 강력하지만, '연산'에는 비용이 더 많이 든다는 점을 기억하자.

저장 공간
JSON은 키 이름도 함께 저장하므로 동일한 데이터라도 일반 컬럼보다 저장 공간을 더 사용한다.

```sql
DROP TABLE IF EXISTS product_flex;
DROP TABLE IF EXISTS product_normal;
```

```sql
-- 1. JSON 구조를 사용하는 테이블 생성
CREATE TABLE product_flex (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    category VARCHAR(50),
    attributes JSON
);

-- 2. 일반적인 정규화 구조를 사용하는 테이블 생성
CREATE TABLE product_normal (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    category VARCHAR(50),
    brand VARCHAR(50),
    price INT,
    stock INT
);

-- 3. 데이터 입력 (JSON 테이블)
INSERT INTO product_flex (category, attributes) VALUES
('스마트폰', '{"brand": "Samsung", "price": 1200000, "stock": 100}');

-- 4. 데이터 입력 (일반 테이블)
INSERT INTO product_normal (category, brand, price, stock) VALUES
('스마트폰', 'Samsung', 1200000, 100);
```

```sql
-- 데이터 크기 비교
SELECT
    'JSON' AS type,
    SUM(LENGTH(attributes)) AS total_bytes
FROM product_flex
WHERE category = '스마트폰'
UNION ALL
SELECT
    'Normal' AS type,
    SUM(LENGTH(brand) + 4 + 4) AS total_bytes  -- brand VARCHAR + price INT +
stock INT
FROM product_normal
WHERE category = '스마트폰'
```

[실행 결과]

type	total_bytes
JSON	52
Normal	15

동일한 데이터를 저장할 때 JSON이 더 많은 공간을 사용한다.

JSON 사용 가이드라인

언제 JSON을 사용하고, 언제 사용하지 말아야 하는지 정리해보자.

JSON을 사용하면 좋은 경우

JSON의 핵심은 유연성이다. 따라서 EAV에서 배운 내용과 동일하다.

카테고리마다 속성이 다른 경우

- 전자제품, 의류, 식품 등 카테고리별로 완전히 다른 속성
- 스마트폰: screen_size, ram, battery
- 의류: size, color, material
- 식품: calories, ingredients, allergens

속성이 자주 변경되거나 추가되는 경우

- 비즈니스 요구사항에 따라 새로운 속성이 계속 추가되는 경우
- 매번 ALTER TABLE을 할 수 없는 상황

스키마를 예측할 수 없는 경우

- 사용자 정의 필드
- 외부 API 응답 저장
- 설정 데이터

데이터 스냅샷 저장

- 주문 시점의 상품 정보
- 이력 데이터

JSON을 사용하면 안 좋은 경우

자주 검색하고 정렬하는 데이터

- 가격으로 자주 정렬하고 필터링한다면 일반 컬럼이 좋다
- `WHERE price BETWEEN 100000 AND 500000 ORDER BY price`

집계와 분석이 필요한 데이터

- 통계, 리포트, 대시보드용 데이터
- SUM, AVG, GROUP BY 등을 자주 사용

참조 무결성이 중요한 데이터

- 다른 테이블과의 관계가 중요한 경우
- 외래 키 제약이 필요한 경우

트랜잭션 무결성이 중요한 데이터

- 금융 데이터
- 재고 수량
- 핵심 비즈니스 데이터

데이터 일관성이 중요한 경우

- 모든 레코드가 동일한 구조를 가져야 하는 경우
- 필수 필드 검증이 필요한 경우

실무 설계 패턴

실무에서 JSON을 효과적으로 활용하는 방법은 하이브리드 방식을 사용하는 것이다.

하이브리드 패턴 (가장 권장)

핵심 속성은 일반 컬럼, 부가 속성은 JSON으로 분리한다.

```
DROP TABLE IF EXISTS product_best_practice;

CREATE TABLE product_best_practice (
```

```sql
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,

    -- 핵심 속성: 일반 컬럼 (검색, 정렬, 집계에 사용)
    name VARCHAR(200) NOT NULL,
    category VARCHAR(100) NOT NULL,
    brand VARCHAR(100) NOT NULL,
    price INT NOT NULL,
    stock INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'active',

    -- 부가 속성: JSON (카테고리별로 다른 속성)
    attributes JSON,

    -- 메타데이터: JSON (SEO, 분석 등)
    metadata JSON,

    created_at DATETIME NOT NULL DEFAULT NOW(),
    updated_at DATETIME NOT NULL DEFAULT NOW() ON UPDATE NOW(),

    -- 일반 컬럼 인덱스
    INDEX idx_category (category),
    INDEX idx_brand (brand),
    INDEX idx_price (price),
    INDEX idx_status (status),
    INDEX idx_category_brand_price (category, brand, price)
);

-- 데이터 입력
INSERT INTO product_best_practice (name, category, brand, price, stock,
attributes, metadata) VALUES
('갤럭시 S24 울트라', '스마트폰', '삼성', 1650000, 100,
 JSON_OBJECT(
     'screen_size', 6.8,
     'ram', 12,
     'storage_options', JSON_ARRAY(256, 512, 1024),
     'colors', JSON_ARRAY('티타늄 블랙', '티타늄 그레이', '티타늄 바이올렛'),
     '5g', true,
     'camera', JSON_OBJECT('main', 200, 'ultra_wide', 12, 'telephoto', 50)
 ),
 JSON_OBJECT(
     'seo', JSON_OBJECT('title', '갤럭시 S24 울트라 최저가', 'keywords',
JSON_ARRAY('갤럭시', 'S24', '울트라')),
     'analytics', JSON_OBJECT('view_count', 0, 'conversion_rate', 0)
```

```
 )),

('나이키 에어맥스 90', '신발', '나이키', 179000, 250,
 JSON_OBJECT(
     'sizes', JSON_ARRAY(250, 255, 260, 265, 270, 275, 280),
     'colors', JSON_ARRAY('화이트', '블랙', '레드'),
     'material', '메쉬/가죽',
     'gender', 'unisex'
 ),
 JSON_OBJECT(
     'seo', JSON_OBJECT('title', '나이키 에어맥스 90 정품', 'keywords',
JSON_ARRAY('나이키', '에어맥스', '운동화')),
     'analytics', JSON_OBJECT('view_count', 0, 'conversion_rate', 0)
 ));
```

```
-- 일반 컬럼으로 빠른 검색
SELECT name, brand, price
FROM product_best_practice
WHERE category = '스마트폰'
  AND brand = '삼성'
  AND price < 2000000
ORDER BY price;
```

[실행 결과]

name	brand	price
갤럭시 S24 울트라	삼성	1650000

```
-- 필요할 때 JSON 속성 조회
SELECT
    name,
    brand,
    price,
    attributes->>'$.screen_size' AS screen_size,
    attributes->>'$.ram' AS ram
FROM product_best_practice
```

```
WHERE category = '스마트폰';
```

[실행 결과]

name	brand	price	screen_size	ram
갤럭시 S24 울트라	삼성	1650000	6.8	12

JSON 설계 정리

지금까지 EAV의 대안으로 JSON 설계를 배우고, 활용하고, 최적화까지 해보았다. 마지막으로 이 설계 방식을 언제 써야 하고, 언제 쓰지 말아야 하는지 정리해보자.

장점

1. **스키마 유연성**: 속성이 자주 변경되거나, 상품마다 속성이 전혀 다른 경우(쇼핑몰 상품 정보, 사용자 설정, 로그 등)에 매우 강력하다.
2. **개발 생산성**: 애플리케이션의 객체(Object)를 DB에 그대로 저장하고 꺼낼 수 있어 매핑 작업이 줄어든다.
3. **조회 단순화**: 데이터를 중첩할 수 있어서, 복잡한 조인(JOIN) 없이 한 번에 데이터를 가져올 수 있다.

단점 및 한계

1. **데이터 무결성 약화**: 외래키(FK) 제약조건을 JSON 내부 값에 걸 수 없다. 예를 들어, JSON 안의 `manufacturer_id`가 실제 `manufacturer` 테이블에 존재하는지 DB가 보장해주지 않는다.
2. **저장 공간 효율**: 데이터 타입이 명시적인 일반 컬럼에 비해, JSON은 키 이름 등을 포함하므로 저장 공간을 더 많이 차지할 수 있다.
3. **데이터 수정의 번거로움**: 특정 속성 하나를 수정하려 해도 `JSON_SET` 등의 함수를 써야 하며, 문법이 일반 UPDATE보다 복잡하다.

결론: 언제 JSON을 써야 할까?

- **메타데이터, 로그, 설정 정보, 비정형 속성 데이터**에는 **JSON**이 적합하다.
- **핵심 비즈니스 데이터, 외래키 참조가 필요한 데이터, 집계와 통계가 빈번한 데이터**는 **전통적인 컬럼 방식**이 적합하다.

JSON을 사용해야 하는 상황이라면, 실무에서는 이 두 가지를 적절히 섞어서 사용하는 **하이브리드 설계**가 정답이다. 우리 쇼핑몰 예제처럼 `name`, `price`, `stock` 같은 핵심 데이터는 일반 컬럼으로, `screen_size`, `material` 같은 가변 속성은 JSON으로 관리하는 것이 가장 이상적인 형태다.

> **항상 전통적인 관계형 데이터베이스 모델링을 고민하자.**
> 그래도 해결이 안되는 부분이 있다면 해당 부분만 JSON을 고려하자.
> 레거시 시스템이 아니라면 EAV 보다는 JSON을 사용하자.

관계형 데이터베이스 vs NoSQL

JSON 설계를 학습하면서 자연스럽게 떠오르는 질문이 있다. "그렇다면 MongoDB 같은 NoSQL을 쓰는 게 더 낫지 않나?" 이 질문에 답하기 위해 관계형 데이터베이스와 NoSQL의 차이를 간단히 비교해보자.

NoSQL 문서 데이터베이스란

MongoDB, CouchDB 같은 문서 데이터베이스는 JSON과 유사한 문서(Document)를 기본 저장 단위로 사용한다.

```
// MongoDB 문서 예시
{
    "_id": ObjectId("507f1f77bcf86cd799439011"),
    "name": "갤럭시 S24",
    "category": "스마트폰",
    "brand": "삼성",
    "price": 1200000,
    "specs": {
        "screen_size": 6.2,
        "ram": 8,
        "storage": 256
    },
    "tags": ["5G", "AI", "플래그십"],
    "created_at": ISODate("2026-01-15T09:00:00Z")
}
```

스키마가 고정되어 있지 않아 유연하게 데이터를 저장할 수 있다.

주요 차이점 비교

데이터 모델

관계형 DB (MySQL)

- 테이블, 행, 열로 구성
- 정규화된 스키마
- 관계(JOIN)로 데이터 연결
- 스키마 변경에 ALTER TABLE 필요

NoSQL (MongoDB)

- 컬렉션, 문서로 구성
- 유연한 스키마 (Schemaless)
- 임베딩 또는 참조로 데이터 연결
- 스키마 변경이 자유로움

트랜잭션과 일관성

관계형 DB (MySQL)

- 강력한 ACID 트랜잭션
- 다중 테이블 트랜잭션 지원
- 강한 일관성 보장

NoSQL (MongoDB)

- 기본적으로 단일 문서 원자성
- 다중 문서 트랜잭션은 제한적으로 지원
- 최종 일관성(Eventual Consistency) 모델

확장성

관계형 DB (MySQL)

- 수직 확장 (Scale-up) 중심
- 수평 확장 (Sharding)은 복잡함
- 읽기 확장은 복제로 가능

NoSQL (MongoDB)

- 수평 확장 (Scale-out) 용이
- 내장된 샤딩 지원
- 분산 환경에 최적화

쿼리 능력

```sql
-- 관계형 DB: 강력한 SQL
SELECT p.name, c.category_name, SUM(o.quantity) as total_sold
FROM products p
JOIN categories c ON p.category_id = c.category_id
JOIN order_items o ON p.product_id = o.product_id
WHERE o.order_date >= '2026-01-01'
GROUP BY p.product_id
HAVING total_sold > 100
ORDER BY total_sold DESC;
```

```javascript
// MongoDB: 집계 파이프라인
db.orders.aggregate([
    { $match: { order_date: { $gte: ISODate("2026-01-01") } } },
    { $unwind: "$items" },
    { $group: { _id: "$items.product_id", total_sold: { $sum:
"$items.quantity" } } },
    { $match: { total_sold: { $gt: 100 } } },
    { $sort: { total_sold: -1 } }
])
```

복잡한 JOIN과 집계는 관계형 DB가 더 강력하다.

언제 무엇을 선택할까

관계형 DB가 좋은 경우

데이터 일관성이 중요한 경우

```sql
-- 금융, 재고, 주문 등 정확성이 중요한 데이터
-- 트랜잭션으로 여러 테이블 동시 업데이트
START TRANSACTION;
UPDATE accounts SET balance = balance - 100000 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100000 WHERE account_id = 2;
```

```
INSERT INTO transactions (from_account, to_account, amount) VALUES (1, 2,
100000);
COMMIT;
```

복잡한 관계와 JOIN이 필요한 경우

```
-- 여러 테이블의 데이터를 결합하는 복잡한 쿼리
SELECT
    o.order_id,
    c.customer_name,
    p.product_name,
    s.shipper_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id
JOIN shippers s ON o.shipper_id = s.shipper_id;
```

기존 관계형 인프라가 있는 경우

- 이미 MySQL/PostgreSQL을 사용 중
- 팀이 SQL에 익숙함
- 기존 도구와 통합 필요

부분적인 스키마 유연성만 필요한 경우

```
-- 핵심 구조는 고정, 일부 속성만 유연하게
-- 이 경우 JSON 컬럼이 적합
CREATE TABLE products (
    product_id BIGINT PRIMARY KEY,
    name VARCHAR(200) NOT NULL,      -- 고정 스키마
    price INT NOT NULL,              -- 고정 스키마
    attributes JSON                  -- 유연한 스키마
);
```

NoSQL이 더 좋은 경우

스키마가 매우 유동적인 경우

- 속성이 수시로 변경됨

- 문서마다 구조가 완전히 다름
- 개발 초기 단계로 스키마가 불확실 (대안: ORM으로 어느 정도 해결 가능)

대규모 수평 확장이 필요한 경우

- 데이터가 TB, PB 단위로 성장
- 여러 서버에 분산 저장 필요
- 지역별 분산 배치 필요

중첩된 문서 구조가 핵심인 경우

```
// 게시글과 댓글을 하나의 문서로
{
    "_id": "post123",
    "title": "게시글 제목",
    "content": "내용...",
    "comments": [
        {"author": "user1", "text": "댓글1", "replies": [...]},
        {"author": "user2", "text": "댓글2", "replies": [...]}
    ]
}
```

하이브리드 아키텍처

실무에서는 하나만 선택하지 않고, 용도에 맞게 여러 데이터베이스를 함께 사용하는 경우가 많다.

예시: 이커머스 시스템

MySQL (관계형)

- 주문, 결제, 재고 (트랜잭션 중요)
- 회원 정보 (일관성 중요)
- 상품 기본 정보 + JSON 속성

MongoDB (문서형)

- 상품 리뷰 (유연한 구조)
- 로그, 이벤트 데이터 (대용량)
- 개인화 추천 데이터

Redis (키-값)

- 세션 저장
- 캐시
- 실시간 순위

Elasticsearch

- 상품 검색
- 로그 분석

실무의 진짜 조언

강의를 마무리하며 솔직한 조언을 하나 하겠다. 기술 블로그나 컨퍼런스에서 NoSQL을 활용한 화려한 아키텍처를 보면, 우리 서비스에도 당장 도입해야 할 것 같은 조급함이 생길 수 있다. 하지만 실무에서 마주하는 프로젝트의 95% 이상은 MySQL이나 PostgreSQL 같은 **관계형 데이터베이스 하나만으로도 충분하다.**

데이터의 구조가 자주 바뀌어서 유연성이 필요하다면 앞서 배웠듯이 관계형 데이터베이스의 **JSON 타입**을 활용하면 된다. 핵심 비즈니스 데이터는 견고한 테이블에 저장하고, 가변적인 속성만 JSON 컬럼에 저장하는 하이브리드 전략을 사용하면 유연성 문제의 대부분이 해결된다.

NoSQL을 섣불리 도입하지 말아야 하는 진짜 이유는 **'운영 복잡도'** 때문이다.
시스템에 새로운 종류의 데이터베이스가 하나 추가된다는 것은 단순히 코드를 조금 더 짜는 문제가 아니다.

- 팀원 모두가 해당 기술을 새로 배워야 한다.
- 모니터링해야 할 서버와 지표가 늘어난다.
- 백업과 복구 전략을 별도로 수립해야 한다.
- 장애가 발생했을 때 트러블슈팅할 수 있는 전문가가 필요하다.

운영하는 시스템이 하나 더 늘어난다는 것은 스타트업이나 소규모 팀에게는 엄청난 부담이다. 잘 아는 RDBMS 하나를 극한까지 튜닝해서 사용하는 것이, 낯선 NoSQL을 어설프게 운영하는 것보다 훨씬 안정적이고 성능도 좋다.

결론은 명확하다. **관계형 데이터베이스로 시작해라.** 그리고 비즈니스가 폭발적으로 성장하여 RDBMS만으로는 도저히 트래픽을 감당할 수 없는 '규모의 경제'가 필요한 시점이 왔을 때, 그때 NoSQL 도입을 검토해도 절대 늦지 않다.

마지막 당부: 관계형 모델을 먼저 생각하라

강의를 마치며 꼭 강조하고 싶은 점이 있다. 우리는 앞서 EAV 패턴의 한계를 극복하기 위해 JSON을 도입했다. 확실히

JSON은 EAV의 복잡한 쿼리 문제를 해결해 주고 개발 편의성을 높여준다.

하지만 **JSON 역시 EAV가 가진 핵심적인 단점은 그대로 안고 있다는 사실**을 잊으면 안 된다. 데이터 무결성을 데이터베이스 레벨에서 강제할 수 없고, 외래 키를 통한 참조 무결성도 보장되지 않는다. 결국 데이터 품질을 유지하는 책임이 온전히 애플리케이션 코드로 넘어가는 것이다. 편의성 뒤에는 언제나 '데이터 품질 저하'라는 위험이 도사리고 있다.

실무에서 데이터 모델링을 할 때의 대원칙은 변하지 않는다.

1. **항상 관계형 데이터베이스 모델(정규화된 테이블)로 문제를 해결할 수 있는지 먼저 치열하게 고민해라.**
2. JSON은 관계형 모델만으로는 도저히 유연성을 감당할 수 없거나, 구현 비용이 비효율적으로 클 때 고려하는 **'차선책'**이자 **'최후의 수단'**이어야 한다.

"JSON을 쓰면 편하니까 대충 묶어서 넣자"라는 유혹에 빠지지 마라. 기본에 충실한 설계가 결국 가장 오래 살아남는 튼튼한 시스템을 만든다. **실무에서는 정말 어쩔 수 없을 때 사용하는 최후의 수단이어야 한다!**

정리

EAV의 한계와 JSON의 필요성

- **EAV 패턴의 한계**: 속성 조회를 위해 `CASE WHEN`, `GROUP BY` 등을 사용한 복잡한 피벗 쿼리가 필요하며, 모든 값이 문자열로 저장되어 데이터 타입 비교 시 형변환 문제가 발생한다. 또한 데이터가 늘어날수록 성능이 급격히 저하된다.
- **JSON의 필요성**: 하나의 컬럼에 다양한 속성을 저장할 수 있어 데이터 구조가 단순해지고, 숫자나 불리언 등 네이티브 데이터 타입을 지원하여 형변환 없이 정확한 비교가 가능하다. 또한 피벗 없이 직관적인 쿼리로 데이터를 추출할 수 있다.
- **JSON 도입 배경**: NoSQL의 유연성을 관계형 데이터베이스에 도입하기 위해 시작되었으며, SQL:2016 표준에 공식 포함되어 주요 RDBMS에서 지원한다.

JSON 문법

- **기본 구조**: 키-값 쌍인 객체(`{}`)와 순서가 있는 목록인 배열(`[]`)로 구성된다.
- **데이터 타입**: 문자열, 숫자(정수/실수), 불리언(`true` / `false`), `null`, 객체, 배열을 지원한다.
- **작성 규칙**: 키(Key)는 반드시 큰따옴표(`""`)로 감싸야 하며, 마지막 요소 뒤에 쉼표를 사용할 수 없다. 또한 주석을 지원하지 않는다.

MySQL에서 JSON 사용하기 1

- **테이블 생성**: 컬럼 타입으로 `JSON`을 지정하여 테이블을 생성한다. 유효한 JSON 형식만 저장되도록 자동 검증된다.
- **데이터 조작**: `JSON_OBJECT`, `JSON_ARRAY` 함수를 사용해 구조화된 데이터를 생성하거나 문자열로 직접 입력할 수 있다.
- **데이터 조회**: `JSON_EXTRACT`(`->`)는 JSON 타입을 반환하고, `JSON_VALUE`(`->>`)는 따옴표를 제거한 문자열을 반환한다.
- **타입 지정 조회**: `JSON_VALUE` 사용 시 `RETURNING` 절을 통해 `UNSIGNED`, `DECIMAL`, `DATE` 등으로 타입을 변환하여 조회할 수 있다.
- **조건 검색**: `WHERE` 절에서 JSON 경로를 통해 숫자, 문자열, 불리언 비교가 가능하며, `JSON_CONTAINS`로 배열 내 특정 값 포함 여부를 확인할 수 있다.

MySQL에서 JSON 사용하기 2

- **데이터 수정 함수**:
  - `JSON_SET`: 키가 있으면 수정, 없으면 추가한다.
  - `JSON_INSERT`: 키가 없을 때만 추가한다.
  - `JSON_REPLACE`: 키가 있을 때만 수정한다.
  - `JSON_REMOVE`: 특정 키를 삭제한다.
  - `JSON_ARRAY_APPEND`: 배열에 요소를 추가한다.
- **유틸리티 함수**: `JSON_KEYS`(키 목록), `JSON_LENGTH`(요소 개수), `JSON_TYPE`(데이터 타입 확인), `JSON_VALID`(유효성 검사), `JSON_PRETTY`(포맷팅) 등을 제공한다.

JSON 활용 - 다양한 실무 사례 1

- **카테고리별 속성 관리**: 공통 속성은 일반 컬럼으로, 카테고리마다 다른 속성은 JSON 컬럼(`attributes`)으로 관리하여 유연성을 확보한다.
- **JSON_TABLE 활용**: JSON 배열 데이터를 관계형 테이블 형식으로 변환하여 조인하거나 집계할 수 있다. 이를 통해 상품 옵션 재고 관리 등의 복잡한 데이터 처리가 쉬워진다.

JSON 활용 - 다양한 실무 사례 2

- **사용자 설정 저장**: 알림 설정, UI 테마 등 개인화된 설정을 JSON 객체로 저장하여 관리한다. `JSON_SET`을 통해 특정 설정만 부분 업데이트할 수 있다.
- **로그 및 이벤트 데이터**: 페이지 조회, 구매, 에러 로그 등 구조가 다양한 이벤트 데이터를 JSON으로 저장하고 분석한다.

JSON 인덱스와 성능 최적화 1

- **성능 문제**: 인덱스 없는 JSON 검색은 풀 테이블 스캔(Full Table Scan)을 유발하여 성능이 매우 느리다.
- **가상 컬럼(Virtual Column) 인덱스**: JSON 내부 경로의 값을 추출하는 가상 컬럼(`GENERATED ALWAYS AS ... VIRTUAL`)을 생성하고, 이에 대해 B-Tree 인덱스를 적용하여 성능을 최적화한다.

- **VIRTUAL vs STORED**: `VIRTUAL`은 메타데이터만 저장하고 읽을 때 계산하며, `STORED`는 물리적으로 저장한다. 인덱싱 목적이라면 저장 공간을 절약하는 `VIRTUAL`이 표준이다.

JSON 인덱스와 성능 최적화 2

- **함수 기반 인덱스**: MySQL 8.0.13부터 가상 컬럼 생성 없이 수식(`(expression)`)에 직접 인덱스를 생성할 수 있다.
- **멀티 밸류 인덱스(Multi-Valued Index)**: MySQL 8.0.17부터 JSON 배열 내부의 요소들을 인덱싱할 수 있다. `CAST(... AS ... ARRAY)`를 사용하며, `MEMBER OF`, `JSON_CONTAINS` 등의 함수와 함께 사용된다.

JSON 설계의 장단점과 한계

- **장점**: 스키마 변경 없이 속성 추가가 가능(유연성), 복잡한 중첩 구조 표현 가능, 애플리케이션 객체와 매핑 용이, EAV 대비 직관적인 쿼리 작성이 가능하다.
- **단점**: 데이터 무결성 및 참조 무결성(FK)을 보장하지 않는다. 집계 및 분석 시 파싱 오버헤드로 성능이 떨어지며, 키 이름을 포함하여 저장하므로 저장 공간을 더 많이 차지한다.
- **제약 조건**: `CHECK` 제약 조건을 통해 일부 유효성 검증은 가능하나 한계가 있다.

JSON 사용 가이드라인

- **사용 권장**: 속성이 자주 변경되거나 예측 불가능한 경우, 메타데이터, 로그, 설정 정보, 데이터 스냅샷 등.
- **사용 지양**: 자주 검색/정렬/집계하는 핵심 데이터, 데이터 무결성이 중요한 금융/재고 데이터, 외래 키 참조가 필요한 경우.
- **하이브리드 패턴**: 핵심 속성(검색, 정렬용)은 일반 컬럼으로, 가변 속성은 JSON으로 분리하여 저장하는 방식이 실무에서 가장 권장된다.

관계형 데이터베이스 vs NoSQL

- **비교**: RDBMS는 데이터 정합성과 복잡한 조인에 강점이 있고, NoSQL(문서형)은 스키마 유연성과 수평 확장에 강점이 있다.
- **실무 조언**: 대부분의 프로젝트는 RDBMS의 JSON 타입 활용(하이브리드 구조)만으로도 충분하다. NoSQL 도입은 운영 복잡도를 증가시키므로, 명확한 확장성 이슈나 비정형 데이터 요구사항이 있을 때 검토하는 것이 좋다.
- **핵심 원칙**: 우선 관계형 모델로 설계를 시도하고, JSON은 유연성이 필수적이거나 구현 비용이 너무 클 때 사용하는 '차선책'으로 접근해야 한다.