9주차

▼ 4. ENUM과 SET

`ENUM`과 `SET`은 문자열 값을 MySQL 내부적으로 숫자 값으로 매핑해 관리하는 타입이다.

4.1 ENUM

테이블의 구조(메타 데이터)에 나열된 목록 중 하나의 값을 가질 수 있는 타입

코드화된 값을 관리하는 것이 가장 큰 용도이다.

`ENUM`은 쿼리에서 `CHAR`나 `VARCHAR` 타입과 같이 문자열처럼 비교하거나 저장할 수 있지만 MySQL 서버가 실제로 값을 저장할 때는 그 값에 매핑된 정숫값을 사용한다. `ENUM` 타입에 사용할 수 있는 최대 아이템 개수는 2^{16}개이다. 아이템의 개수가 255개 미만이면 저장 공간으로 1바이트를 사용하고, 그 이상인 경우는 2바이트까지 사용한다.

보통 정숫값은 테이블 정의에 나열된 순서대로 1부터 할당되며 빈 문자열은 항상 0으로 매핑된다.

5.6 버전 이전에는 `ENUM` 타입에 새로운 아이템을 추가해야 하면 항상 테이블을 리빌드해야 했지만, 5.6부터는 새로운 아이템이 `ENUM` 타입 가장 마지막에 추가되는 형태라면 메타데이터 변경만으로 즉시 완료된다.

```
CREATE TABLE tb_enum
(
    fd_enum ENUM ('A', 'B', 'C')
);

ALTER TABLE tb_enum
    MODIFY fd_enum ENUM ('A', 'B', 'C', 'D'),
    ALGORITHM = INSTANT;

ALTER TABLE tb_enum
    MODIFY fd_enum ENUM ('A', 'D','B', 'C'),
    ALGORITHM = COPY,
    LOCK = SHARED;
```

만약 `ENUM` 타입의 문자열 값으로 강제 정렬(기본적으로 정숫값으로 정렬 됨)하려면 `CAST()` 함수를 이용해 문자열 타입으로 변환해 정렬해야 한다.(인덱스 사용 불가)

`ENUM`의 장점은 정의된 코드값만을 사용하도록 강제하는 것 외에도 디스크 저장 공간을 절약할 수 있다는 점이다. (레코드 건수가 매우 많은 경우 더 효과적) 또한 백업이나 복구 시간도 줄일 수 있다. + 스키마 변경 시간과 인덱스 생성 시간 감소

4.2 SET

`ENUM`과 마찬가지로 테이블 구조에 정의된 아이템을 정숫값으로 매핑해 저장하는 방식이다. `ENUM`과의 가장 큰 차이는 `SET`은 하나의 컬럼에 1개 이상의 값을 저장할 수 있다.

실제 여러 개의 값을 저장하는 공간을 가지는 것이 아닌 내부적으로 BIT-OR 연산을 거쳐 1개 이상의 선택된 값을 저장한다. 그래서 각 아이템 값에 매핑되는 정숫값은 1씩 증가하는 정숫값이 아닌 2n의 값을 갖게 된다. 아이템 값의 멤버 수가 8개 이하이면 1바이트의 저장 공간을 사용하고, 9 ~ 16개는 2바이트, ... 최대 8바이트까지(64개) 저장 공간을 사용한다.

여러 값을 컬럼에 저장하려면 `'A,B'`와 같이 콤마로 구분해 문자열 값을 나열해 입력하면 된다. 쿼리 결과에서 반환할 때에도 콤마로 구분해 연결된 문자열을 반환한다. `SET` 타입의 컬럼에서 특정 아이템 멤버를 가진 레코드를 검색하려면 `FIND_IN_SET()` 함수나 `LIKE` 검색을 이용할 수 있다.

```
SELECT *
FROM tb_set
WHERE FIND_IN_SET('A', fd_set);

SELECT *
FROM tb_set
WHERE fd_set LIKE '%A%'; // CA와 같은 아이템이 검색될 수 있음
```

`SET` 타입 컬럼에 동등 비교를 수행하려면 컬럼에 저장된 순서대로 문자열을 나열해야만 검색할 수 있다. `SET` 타입 컬럼에 인덱스가 있더라도 동등 비교 조건을 제외하고 `FIND_IN_SET()` 함수나 `LIKE`를 사용하는 쿼리는 인덱스를 사용할 수 없다.

```
SELECT *
FROM tb_set
WHERE FIND_IN_SET('A', fd_set) >=2;
```

`FIND_IN_SET()` 함수는 첫 번째 인자의 값이 두 번째 인자 컬럼에서 포함된 순서(ex. `FIND_IN_SET('B', 'A,B,C')`은 2를 반환)를 반환한다. 위의 쿼리는 `A`가 두 번째 이후에 포함되는 레코드만 조회한다. 이러한 쿼리는 인덱스를 효율적으로 이용할 수 없으므로 이런 검색이 빈번히 사용된다면 `SET` 타입의 컬럼을 정규화해서 별도로 인덱스를 가진 자식 테이블을 생성하는 것이 좋다.

`ENUM`과 마찬가지로 `SET` 타입 또한 새로운 아이템을 중간에 추가하는 경우 테이블의 읽기 잠금과 리빌드 작업이 필요하다. 가장 마지막에 추가하는 작업은 `INSTANT` 알고리즘으로 메타 정보만 변경하고 즉시 완료되지만, 아이템 개수가 8개를 넘어서게 되는 경우 읽기 잠금과 테이블 리빌드 작업이 필요하다.(저장 공간이 1바이트에서 2바이트로 변경되어야 하기 때문)

▼ 5. TEXT와 BLOB

대량의 데이터를 저장하기 위한 타입

유일한 차이점

`TEXT` 타입은 문자열을 저장하는 대용량 컬럼이라 문자 집합이나 콜레이션을 갖지만, `BLOB` 타입은 이진 데이터 타입이라 별도의 문자 집합이나 콜레이션을 가지지 않는다.

데이터 타입	필요 저장 공간 (L = 저장 데이터 바이트 수)	저장 가능한 최대 바이트 수
`TINYTEXT`, `TINYBLOB`	L + 1바이트	$2^8 - 1$ (255)
`TEXT`, `BLOB`	L + 2바이트	$2^{16} - 1$ (65,535)
`MEDIUMTEXT`, `MEDIUMBLOB`	L + 3바이트	$2^{24} - 1$ (16,777,215)
`LONGTEXT`, `LONGBLOB`	L + 4바이트	$2^{32} - 1$ (4,294,967,295)

`LONG`이나 `LONG VARCHAR` 타입도 있지만 MEDIUMTEXT의 동의어이므로 기억할 필요는 없다.

이진 데이터 타입과 문자열 데이터 타입

	고정길이	가변길이	대용량
문자 데이터	`CHAR`	`VARCHAR`	`TEXT`
이진 데이터	`BINARY`	`VARBINARY`	`BLOB`

TEXT와 BLOB 타입이 적합한 상황

- 컬럼 하나에 저장되는 문자열이나 이진 값의 길이가 예측 불가능할 정도로 클 때
- 레코드 전체 크기가 64KB를 넘어서서 더 큰 컬럼을 추가할 수 없는 경우

> 다른 DBMS와 달리 MySQL은 값의 크기가 4,000바이트를 넘을 때 반드시 `BLOB`이나 `TEXT`를 사용해야 하는 것은 아니다. MySQL에서는 레코드의 전체 크기가 64KB를 넘지 않는 한도 내에서는 `VARCHAR`나 `VARBINARY`의 길이 제한이 없다.

MySQL에서 인덱스 레코드의 모든 컬럼은 최대 제한 크기를 가진다.

- MyISAM: 1,000바이트
- `REDUNDANT` 또는 `COMPACT` 로우 포맷 InnoDB: 767바이트
- `DYNAMIC` 또는 `COMPRESSED` 로우 포맷 InnoDB: 3,072바이트

`BLOB`이나 `TEXT` 타입 컬럼에 인덱스를 생성할 때는 컬럼값의 몇 바이트까지 인덱스를 생성할 것인지 명시해야 할 때도 있다.(최대 크기를 초과하는 인덱스는 생성 불가능)

`BLOB`이나 `TEXT` 컬럼으로 정렬을 수행할 때 저장된 값이 10MB라고 하더라도 실제로는 `max_sort_length` 시스템 변수 값 길이까지만 정렬을 수행한다. 만약 `TEXT` 타입 정렬을 더 빠르게 수행하고 싶으면 이 값을 줄여 설정하는 것이 좋다.

쿼리 특성에 따라 임시 테이블이 생성될 때가 있는데 임시 테이블이 메모리에 저장될 때는 `internal_tmp_meme_storage_engine` 시스템 변수 설정에 따라 MEMORY 스토리지 엔진이나 TempTable 스토리지 엔진 중 하나를 사용한다. 8.0 버전부터 TempTable은 `TEXT`나 `BLOB` 타입을 지원하지만 MEMORY 스토리지 엔진은 지원하지 않는다. 가능하면 시스템 변수를 `TempTable`로 설정하자.

`BLOB`이나 `TEXT` 컬럼에 포함된 테이블의 `INSERT`, `UPDATE` 쿼리는 매우 길어질 수 있는데 `max_allowed_packet` 시스템 변수에 설정된 값보다 큰 SQL 문장은 서버로 전송되지 못하고 오류가 발생할 수 있으니 대용량 `BLOB`, `TEXT` 컬

럼을 사용하는 쿼리가 있으면 시스템 변수 값을 충분히 늘려 설정하는 것이 좋다.

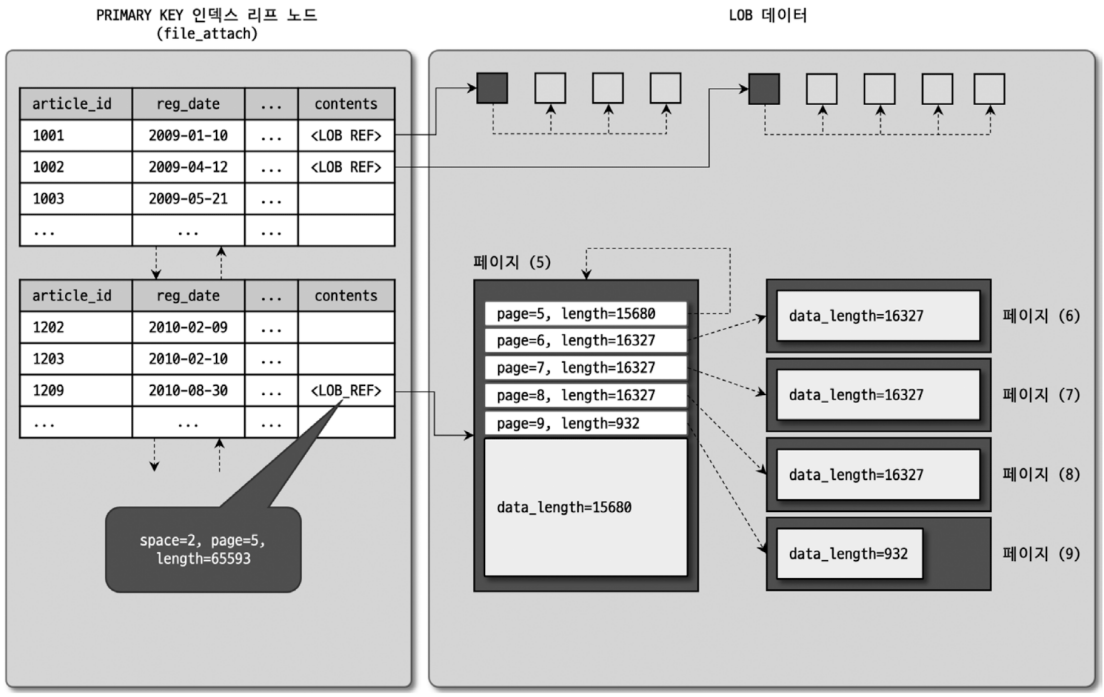
`BLOB`, `TEXT` 타입 컬럼의 데이터가 저장되는 방식을 결정하는 요소는 테이블의 `ROW_FORMAT` 옵션이다. 테이블을 생성할 때 해당 옵션을 별도로 지정하지 않으면 `innodb_default_row_format` 시스템 변수에 설정된 값을 적용한다. 기본적으로 최신 `ROW_FORMAT`인 `dynamic`이 설정된다.

5.6 버전에서 테이블의 기본 `ROW_FORMAT`은 `COMPACT`, 5.7 부터는 `DYNAMIC`. `COMPACT`는 나머지 모든 `ROW_FORMAT`의 바탕이 되는 포맷이다. `COMPACT` 포맷에서 저장 가능한 레코드 하나의 최대 길이는 데이터 페이지(데이터 블록) 크기 16KB의 절반인 8,126바이트이다.(데이터 페이지에서 관리용으로 사용되는 공간을 제외하고 사용 가능한 최대 공간의 절반) 이 경우 `BLOB`이나 `TEXT` 타입의 컬럼을 가능한 레코드에 같이 포함해 저장하려고 할 것이다. 만약 레코드 전체 길이가 8,126바이트를 넘어서면 용량이 큰 컬럼 순서대로 외부 페이지(Off-page or External-page)로 옮길 것이다.

BLOB 타입 컬럼과 TEXT 타입 컬럼을 모두 가진 테이블의 예시

`BLOB` 컬럼 길이	`TEXT` 컬럼 길이	`BLOB` 컬럼 저장 위치	`TEXT` 컬럼 저장 위치
3,000	3,000	PK 페이지	PK 페이지
3,000	10,000	PK 페이지	외부 페이지
10,000	10,000	외부 페이지	외부 페이지

`BLOB`이나 `TEXT` 컬럼이 외부 페이지로 저장될 때 길이가 16KB를 넘는 경우 컬럼의 값을 나누어 여러 개의 외부 페이지에 저장하고 각 페이지를 체인으로 연결한다.



BLOB이나 TEXT 컬럼 값이 여러 개의 외부 페이지로 저장된 형태

`COMPACT`와 `REDUNDANT` `ROW_FORMAT`을 사용하는 테이블에서는 외부 페이지로 저장된 `BLOB`, `TEXT` 컬럼의 앞쪽 768바이트(BLOB prefix)만 잘라 PK 페이지에 같이 저장한다. `DYNAMIC`이나 `COMPRESSED` 포맷에서는 BLOB prefix를 PK

페이지에 저장하지 않는다. `COMPACT`나 `REDUNDANT` 포맷의 BLOB prefix는 인덱스를 생성할 때 도움이 되지만 `BLOB`이나 `TEXT`를 가진 테이블의 저장 효율을 낮추게 될 수 있다. BLOB prefix는 PK 페이지에 저장할 수 있는 레코드 건수를 줄이지만, `BLOB`이나 `TEXT` 컬럼을 거의 참조하지 않는 쿼리는 성능이 더 떨어진다.

▼ 6. 공간 데이터 타입

- `POINT`: 하나의 점
- `LINESTRING`: 하나의 라인
- `POLYGON`: 하나의 다각형
- `GEOMETRY`: `POINT`, `LINESTRING`, `POLYGON`의 슈퍼 타입(모두 저장 가능)

- `MULTIPOINT`: 여러 개의 `POINT`
- `MULTILINESTRING`: 여러 개의 `LINESTRING`
- `MULTIPOLYGON`: 여러 개의 `POLYGON`
- `GEOMETRYCOLLETION`: `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`의 슈퍼 타입(모두 저장 가능)

근데 보통 공간 데이터는 대부분 `POINT`와 `POLYGON` 타입으로 충분하다.

`GEOMETRY` 타입과 모든 자식 타입은 MySQL 서버 메모리에서 관리할 때와 클라이언트로 전송할 때, 디스크에 저장할 때 모두 `BLOB` 객체이다.(`GEOMETRY`가 `BLOB`을 감싸고 있는 구조)

JDBC 표준에서 아직 공간 데이터를 공식적으로 지원하지 않아 MySQLConnector/J(JDBC 드라이버)만으로는 `POINT`나 `POLYGON` 같은 자바 클래스를 사용할 수 없다. 그래서 ORM 라이브러리들은 JTS와 같은 오픈소스 공간 데이터 라이브러리를 사용한다. `GEOMETRY` 컬럼에 저장된 데이터가 일관되고 간단하면 공간 함수(`ST_AsText()`, `ST_X()`, `ST_Y()` 등)를 이용해 JDBC에서 지원하는 데이터 타입으로 변환 후 조회하는 방법도 있다.

6.1 공간 데이터 생성

`POINT` 타입

- WKT: `POINT(x y)`
- 객체 생성: `ST_PointFromText('${WKT}')`

`LINESTRING` 타입

- WKT: `LINESTRING(x0 y0, x1 y1, x2 y2, x3 y3, …)`
- 객체 생성: `ST_LineStringFromText('${WKT}')`

`PLYGON` 타입

- WKT: `POLYGON((x0 y0, x1 y1, x2 y2, x3 y3, x0 y0))`
- 객체 생성: `ST_PloygonFromText('${WKT}')`

`MULTIPOINT` 타입

- WKT: `MULTIPOINT(x0 y0, x1 y1, x2 y2)`
- 객체 생성: `ST_MultiPointFromText('${WKT}')`

`MULTILINESTRING` 타입

- WKT: `MULTILINESTRING((x0 y0, x1 y1), (x2 y2, x3 y3))`
- 객체 생성: `ST_MultiLineStringFromText('${WKT}')`

`MULTIPOLYGON` 타입

- WKT: `MULTIPOLYGON(((x0 y0, x1 y1, x2 y2, x3 y3, x0 y0)), ((x4 y4, x5 y5, x6 y6, x7 y7, x4 y4)))`
- 객체 생성: `ST_MultiPolygonFromText('${WKT}')`s

`GEOMETRYCOLLECTION` 타입

- WKT: `GEOMETRYCOLLECTION(POINT(x0 y0), POINT(x1 y1), LINESTRING(x2 y2, x3 y3))`
- 객체 생성: `ST_GeometryCollectionFromText('${WKT}')`

각 공간 데이터 생성 함수 이름에서 `FromText` 대신 `FromWKB`를 사용하면 WKT 대신 WKB를 이용한 공간 데이터 객체를 생성할 수 있다.

> 예전 버전에서는 `POINT(x y)`, `GeomFromText('POINT(x y)')`으로 `POINT` 객체를 생성할 수 있었지만(나머지 공간 데이터 타입도) 이는 OpenGIS 표준이 아닌 MySQL 자체 방법이었기 때문에 8.0 이후부터 OpenGIS 표준을 준수하기 위해 비표준 함수들은 언제든 제거될 수 있다. `ST_` 접두사 가진 함수들을 우선 사용하자.

위의 공간 데이터 생성 함수들은 모두 첫 번째 파라미터로 WKT를 사용하고 두 번재 파라미터로 SRID를 설정할 수 있다.(SRID를 명시하지 않으면 기본 0)

6.2 공간 데이터 조회

- `ST_AsText()` / `ST_AsWKT()`: WKB 포맷 조회
- `ST_AsBinary()` / `ST_AsWKB()`: WKT 포맷 조회
- 그냥 조회: MySQL 이진 포맷
- 공간 데이터의 속성 함수를 이용한 조회

MySQL :: MySQL 8.0 Reference Manual :: 14.16.7.2 Point Property Functions

A Point consists of X and Y coordinates,
which may be obtained using the
ST_X() and

https://dev.mysql.com/doc/refman/8.0/en/gis-point-property-functions.html

MySQL :: MySQL 8.0 Reference Manual :: 14.16.7.3 LineString and MultiLineString Property Functions

A LineString consists of
Point values. You can extract particular
points of a LineString, count the number of

https://dev.mysql.com/doc/refman/8.0/en/gis-linestring-property-functions.html

`ST_Length()` 함수는 라인 전체 길이를 반환하는데 SRID 값에 따라 계산 방식이 달라질 수 있다. SRID 4326 좌표계를 사용 시 `ST_Length()`는 구면체를 가정하지 않은 결과이기 때문에 구체면상 거리는 `ST_Distance_Sphere()` 함수를 사용해야 한다.

> MySQL :: MySQL 8.0 Reference Manual :: 14.16.7.4 Polygon and MultiPolygon Property Functions
> Functions in this section return properties of
> Polygon or MultiPolygon
> values.
> https://dev.mysql.com/doc/refman/8.0/en/gis-polygon-property-functions.html

▼ 7. JSON 타입

5.7 버전부터 `JSON` 타입이 지원되기 시작하였고, 8.0 버전으로 업그레이드되면서 많은 기능과 성능 개선 사항이 추가되었다. `TEXT`나 `BLOB` 타입으로 저장해도 되지만, `JSON` 타입은 MongoDB와 같이 바이너리 포맷의 `BSON`(Binary JSON)으로 변환해 저장한다.

7.1 저장 방식

내부적으로 `JSON` 타입의 값을 `BLOB`에 저장하지만 사용자가 입력한 값 그대로 저장하는 것이 아닌 `BSON` 타입으로 변환해 저장한다. ⇒ 공간 효율이 높다.

```
CREATE TABLE tb_json
(
    id INT,
    fd JSON
);

INSERT INTO tb_json
VALUES (1,
        '{
          "user_id": 1234567890
        }'),
       (2,
        '{
          "user_id": "1234567890"
        }');

SELECT id, fd, JSON_TYPE(fd -> "$.user_id") AS field_type, JSON_STORAGE_SIZE(
FROM tb_json;
```

	id	fd	field_type	byte_size
1	1	{"user_id": 1234567890}	INTEGER	23
2	2	{"user_id": "1234567890"}	STRING	30

첫 번째 레코드는 `user_id` 필드 값을 정수 타입으로, 두 번째 레코드는 문자열 타입으로 저장했다. 그 결과 두 레코드의 JSON 값을 이진 포맷으로 변환했을 때 7바이트의 공간 차이가 발생한다.

이진 포맷 JSON 데이터에는 JSON 도큐먼트를 구성하는 모든 키의 위치와 키의 이름이 각 JSON 필드의 값보다 먼저 나열되어 있기 때문에 `JSON` 컬럼의 특정 필드만 참조하거나 특정 필드의 값만 업데이트(길이가 변경되지 않는 부분 업데이트)하는 경우 `JSON` 컬럼의 값을 모두 읽지 않고도 즉시 원하는 필드의 이름을 읽거나 변경할 수 있다.

매우 큰 JSON 도큐먼트가 저장되면 16KB 단위로 여러 개의 데이터 페이지로 나누어 저장한다. 이때 `BLOB` 페이지로 나누어 저장하는데 5.7 버전까지는 `BLOB` 페이지들이 단순 링크드 리스트(Linked List)처럼 관리되었다. 하지만 이 형태는 JSON 필드의 부분 업데이트를 효율적으로 처리할 수 없기 때문에 8.0 버전부터 `BLOB` 페이지들의 인덱스를 관리하고 각 인덱스는 실제 BLOB 데이터를 가진 페이지들의 링크를 갖도록 개선되었다.

7.2 부분 업데이트 성능

`JSON` 컬럼의 부분 업데이트 기능은 `JSON_SET()`과 `JSON_REPLACE()`, `JSON_REMOVE()` 함수를 이용해 JSON 도큐먼트의 특정 필드 값을 변경하거나 삭제하는 경우에만 동작한다.

```
UPDATE tb_json
SET fd = JSON_SET(fd, '$.user_id', "12345")
WHERE id = 2;
```

`JSON` 컬럼 값 변경 작업이 '부분 업데이트'로 처리되었는지 확인할 수 있는 명확한 방법은 없고, `JSON_STORAGE_SIZE()` 함수와 `JSON_STORAGE_FREE()` 함수를 이용해 대략적인 예측이 가능하다.

`JSON` 컬럼의 전체 업데이트와 부분 업데이트의 성능 차이는 크게 느껴지지 않을 수 있는데, 부분 업데이트 기능은 특정 조건에서는 매우 빠른 업데이트 성능을 보여준다. `JSON` 컬럼은 내부적으로 정확히는 `LONGBLOB` 타입으로 저장되는데 최대 4GB까지 값을 가질 수 있다.
1MB JSON 데이터를 저장해도 16KB 데이터 페이지를 64개나 사용하는데 부분 업데이트의 경우 이 중 하나만 변경하면 되지만 그게 아니라면 64개 데이터 페이지를 다시 디스크로 기록해야 한다.

MySQL 서버에서는 일반적으로 복제를 사용하기 때문에 JSON 변경 내용을 바이너리 로그에 기록해야 한다. 이때 여전히 JSON의 데이터를 모두 기록한다. 변경된 내용들만 바이너리 로그에 기록되도록 `binlog_row_value_options`과 `binlog_row_image` 시스템 변수 설정값을 변경하면 `JSON` 컬럼의 부분 업데이트 성능을 훨씬 빠르게 만들 수 있다.

```
SET binlog_format = ROW; // STATEMENT 타입으로 설정해도 거의 동일한 성능 향상
SET binlog_row_value_options = PARTIAL_JSON;
SET binlog_row_image = MINIMAL;
```

> 테이블에 PK가 없는 경우에는 부분 업데이트 최적화 효과를 얻을 수 없다. PK가 없다면, 복제 시 업데이트할 레코드를 식별하기 위해 레코드의 모든 컬럼을 필요로 한다.

단순히 정수 필드 값을 변경하는 `UPDATE`는 항상 부분 업데이트 기능이 적용된다.

7.3 JSON 타입 콜레이션과 비교

`JSON` 컬럼에 저장되는 데이터와 `JSON`컬럼으로부터 가공되어 나온 결괏값은 모두 `utf8mb4` 문자 집합과 `utf8mb4_bin` 콜레이션을 가진다. 이는 바이너리 콜레이션이기 때문에 `JSON` 컬럼의 비교와 `JSON` 컬럼으로부터 가공된 문자열은 대소문자 구분과 액센트 문자 등을 구분해 비교한다.

7.4 JSON 컬럼 선택

성능을 중심으로 판단하면 `JSON` 컬럼보다는 성능적인 이점을 가지는 정규화된 컬럼을 추천한다. 정규화된 컬럼은 컬럼 이름을 메타 정보로만 저장하지만 `JSON` 컬럼은 각 필드의 이름이 데이터 파일에 매번 저장되어야 하기 때문에 레코드 건수가 많아질수록 필드 이름이 차지하는 디스크 공간이 더욱 커질 것이다.

압축을 사용하면 디스크 공간은 줄일 수 있지만 메모리 사용 효율을 높이지는 못한다. 그리고 MySQL의 압축은 다른 DBMS와 달리 메모리에 압축된 페이지와 압축 해제된 페이지가 공존해야 하기 때문에 메모리 및 CPU 효율을 저하시킬 수 있다.

`JSON` 컬럼이 가지는 장점은 만약 레코드가 가지는 중요도가 낮고 선택적인 속성이 있다면 `JSON` 컬럼을 만들어 저장하면 좋다.(중요도가 낮다는 것은 검색 조건으로 사용될 가능성도 낮고, 쿼리에 자주 접근될 가능성도 낮은 경우) 또한 너무 정규화된 테이블 구조를 유지하면 테이블의 개수가 많아지고 응용 프로그램의 코드도 길어지는 경우가 많다.

> 대부분의 RDBMS는 작은 크기의 데이터 처리에 적합하도록 설계되었다. 그래서 `JSON` 컬럼에 큰 값을 저장하면 느린 성능을 보이기도 한다. 따라서 `JSON` 컬럼을 남용하거나 너무 큰 데이터를 저장하는 것은 권장하지 않는다.

▼ 8. 가상 컬럼(파생 컬럼)

다른 DBMS에서는 “가상 컬럼(Virtual Column)”이라는 이름으로 사용되지만 MySQL 서버에서는 “Generated Column”이라는 이름으로 소개된다.

MySQL 서버의 가상 컬럼은 가상(Virtual) 컬럼과 스토어드(Stored) 컬럼으로 구분할 수 있다.

```
CREATE TABLE tb_virtual_column
(
    id          INT            NOT NULL AUTO_INCREMENT,
    price       DECIMAL(10, 2) NOT NULL DEFAULT '0.00',
    quantity    INT            NOT NULL DEFAULT 1,
    total_price DECIMAL(10, 2) AS (quantity * price) VIRTUAL,
    PRIMARY KEY (id)
);
```

```
CREATE TABLE tb_stored_column
(
    id          INT            NOT NULL AUTO_INCREMENT,
    price       DECIMAL(10, 2) NOT NULL DEFAULT '0.00',
    quantity    INT            NOT NULL DEFAULT 1,
    total_price DECIMAL(10, 2) AS (quantity * price) STORED,
```

```
    PRIMARY KEY (id)
);
```

두 컬럼 모두 컬럼의 정의 뒤에 `AS` 절로 계산식을 정의한다. `VIRTUAL` 또는 `STORED` 키워드를 명시하며 만약 아무것도 명시하지 않으면 `VIRTUAL`로 생성된다. 가상 컬럼은 다른 컬럼의 값을 참조해 계산된 값을 관리하기 때문에 항상 계산식이나 데이터 가공을 위한 표현식을 `AS` 뒤에 정의해야 한다.

가상 컬럼의 표현식은 동일한 입력에 대해 결과가 항상 동일한(DETERMINISTIC) 표현식만 사용 가능하다. 사용자 변수나 NON-DETERMINISTIC 옵션의 함수와 표현식은 사용할 수 없다. 그리고 8.0 버전까지는 가상 컬럼의 표현식에 서브쿼리나 스토어드 프로그램(스토어드 프로시저나 스토어드 함수)을 사용할 수 없다.

가상 컬럼과 스토어드 컬럼의 차이점

- 가상 컬럼(Virtual Column)
  - 컬럼의 값이 디스크에 저장되지 않음
  - 컬럼의 구조 변경은 테이블 리빌드 불필요
  - 컬럼의 값은 레코드가 읽히기 전 또는 BEFORE 트리거 실행 직후에 계산되어 생성
- 스토어드 컬럼(Stored Column)
  - 컬럼의 값이 물리적으로 디스크에 저장
  - 컬럼의 구조 변경은 다른 일반 테이블과 같이 필요 시 테이블 리빌드 방식으로 처리
  - `INSERT`와 `UPDATE` 시점에만 컬럼 값 계산

가상 컬럼이 항상 디스크에 저장되지 않는 것은 아니다. 가상 컬럼에 인덱스를 생성하면 테이블의 레코드는 가상 컬럼을 포함하지 않지만 해당 인덱스는 계산된 값을 저장한다. 그래서 인덱스가 생성된 가상 컬럼의 경우 변경이 필요하면 인덱스의 리빌드 작업이 필요하다.

8.0 버전부터 도입된 함수 기반 인덱스(Function Based Index)는 가상 컬럼에 인덱스를 생성하는 방식으로 동작한다. 함수 기반 인덱스는 테이블을 조회하면 가상 컬럼이 결과에 표시되지 않는다는 차이가 있지만 내부적으로 함수 기반 인덱스와 가상 컬럼은 동일한 방식으로 처리된다.

값을 계산하는 과정이 복잡하고 오래걸리면 스토어드 컬럼, 계산 과정이 빠른 반면 상대적으로 많은 저장 공간을 차지한다면 가상 컬럼을 선택하는 것이 좋다. 알아서 잘 선택하기