[9주차] 15.4 ~ 16전

태그	Done
날짜	@2024년 3월 26일 → 2024년 4월 2일
제목	데이터 타입

📌 15장 데이터 타입

✅ 15.4 ENUM과 SET

- ENUM, SET은 모두 문자열 값을 숫자 값으로 `MySQL 내부적으로 매핑하여 관리` 합니다.

15.4.1 ENUM

- ENUM 타입은 테이블의 구조(메타 데이터)에 나열된 목록 중 하나의 값을 가집니다.
- 코드화된 값을 관리하는게 가장 큰 용도입니다.
- ENUM 타입은 문자열 처럼 비교, 저장이 가능하지만 **실제 디스크 메모리 저장에는 매핑된 정숫값을 사용**합니다.
- ENUM 최대 아이템 개수는 `2^16 - 1개`, 255개 미만이면 저장 공간 1바이트를 이상이면 2바이트까지 사용)

```
# ENUM 생성
CREATE TABLE tb_enum
(
    fd_enum ENUM ('PROCESSING', 'FAILURE', 'SUCCESS')
);

INSERT INTO tb_enum
VALUES ('PROCESSING'), # 내부적으로 1, 2 값으로 순차적으로 매핑됨
       ('FAILURE');

SELECT *
FROM tb_enum;

# 내부적으로 저장된 숫자 값으로 연산이 실행됨
SELECT fd_enum * 1 AS fd_enum_real_value
FROM tb_enum;

# 숫자값으로 검색 가능
SELECT *
FROM tb_enum
WHERE fd_enum = 1;

# VARCHAR처럼 문자로도 검색이 가능함
SELECT *
FROM tb_enum
WHERE fd_enum = 'PROCESSING';
```

- 일반적으로 문자열 순서대로 `1`부터 할당되지만, `""`은 항상 `0으로 매핑`됩니다.
- 별도의 코드 테이블을 사용하는대신 유용하게 사용할 수 있음(주문번호, 주문상태 등)

5.6 이전까지의 단점, 5.6부터의 보완

- 5.6이전에는 문자열 값이 테이블의 구조(메타정보)가 되면서 새로운 ENUM값이 추가되면 테이블을 리빌딩함
- 하지만 5.6부턴 새로 추가하는 `ENUM타입의 제일 마지막으로 추가되는 형태` 일 경우 테이블의 구조(메타데이터) 변경만으로 즉시 완료

```
mysql> ALTER TABLE tb_enum
       MODIFY fd_enum ENUM('PROCESSING','FAILURE','SUCCESS','REFUND'),
         ALGORITHM=INSTANT;

mysql> ALTER TABLE tb_enum
       MODIFY fd_enum ENUM('PROCESSING','FAILURE','REFUND','SUCCESS'),
         ALGORITHM=COPY, LOCK=SHARED;
```

- 기존 순서를 지키고, 맨 뒤에 추가해야 합니다. → mysql 서버의 가용성을 높이는 방법

ENUM 타입 정렬하기

ENUM은 코드화된 칼럼을 mysql이 자제적으로 제공하는 기능입니다. 따라서 정렬을 하면 매핑된 코드값(정수)를 기준으로 정렬합니다.

문자열 기준으로 정렬하고 싶다면 두 가지 방법이 존재합니다.

1. **테이블 생성시 원하는 정렬상태로 생성하기**
2. 문자열로 캐스팅해 정렬하기 (`인덱스 사용 불가`)

ENUM의 장점

1. 테이블 구조에 정의된 코드 값만 사용하는걸 강제함
2. DB서버의 디스크 저장 공간의 크기를 줄여줌
   - 하드웨어 용량이 커도 디스크 → InnoDB 버퍼풀로 적재돼야 쿼리에서 사용가능, 따라서 ENUM을 통해 디스크 저장 공간의 크기를 줄이는 것은 엄청난장점이 됩니다.
   - 디스크 크기와 백업, 복구, 스키마 변경, 인덱스 생성 시간도 반비례하므로 크기는 중요

15.4.2 SET

- 정숫값으로 매핑하여 저장하는 방식은 ENUM과 동일함

- SET은 하나의 칼럼에 1개 이상의 값을 저장할 수 있음
  (mysql 내부의 BIT-OR연산을 통해 1개 이상의 선택된 값을 저장함)

```
# SET
CREATE TABLE tb_set
(
    fd_set SET ('TENNIS','SOCCER','GOLF','TABLE-TENNIS','BASK
);

# 여러개의 값 저장 가능
INSERT INTO tb_set (fd_set)
VALUES ('SOCCER'),
       ('GOLF,TENNIS');

SELECT *
FROM tb_set;
```

- 여러개 저장 가능하지만, 여거개의 값을 저장하는 공간을 갖진 않습니다.
- 매핑되는 정숫값은 2N의 값을 갖습니다.
  - 각 아이템 값의 멤버수가 8개면 1byte, 9 ~ 16개면 2바이트를 사용하고 최대 8바이트까지 사용합니다.
  - `('TENNIS','SOCCER','GOLF','TABLE-TENNIS','BASKETBALL','BILLIARD')` 의 경우 1Byte 8개의 비트에서 LSB부터 1비트씩 채워집니다. → `00111111(2)`

SET 검색 및 인덱스 사용 여부

- **(인덱스 사용 불가)** 저장된 SET을 검색할때는 `FIND_IN_SET()` 혹은 `LIKE 검색`을 사용 할 수 있습니다.
  - 빈번히 사용된다면 SET 타입 칼럼을 정규화해 별도의 인덱스를 가진 자식 테이블을 생성하는게 좋습니다.

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

SELECT *
```

```
FROM tb_set
WHERE fd_set LIKE '%GOLF%';

# CREATE TABLE시 저장된 SET의 순서대로 번호가 매겨짐, 따라서 해당 번
# TENNIS는 첫번째에 위치하므로 1을 반환함
# 따라서 모든 레코드에 대해 TENNIS를 포함하고 있는지 확인할 수 있음
SELECT *
FROM tb_set
WHERE FIND_IN_SET('TENNIS', fd_set) >= 1;
```

- **(인덱스 사용 가능)** 동등 비교는 칼럼에 `저장된 순서대로 문자열을 나열` 해야만 검색가능

```
SELECT *
FROM tb_set
WHERE fd_set = 'TENNIS,GOLF';

SELECT *
FROM tb_set
WHERE fd_set = 'GOLF,TENNIS';
```

SET 타입 추가에 따른 잠금과 리빌딩 (ENUM과 비슷 하지만 조금 다름)

- SET 타입에 정의된 아이템 중간에 새로운 아이템을 추가하면 읽기 잠금 및 리빌딩 작업을 함
- 마지막에 추가하는 겨우 ENUM과 동일하지만 아이템 개수가 9개 이상이 된다면 2바이트가 필요하므로 역시 테이블 리빌딩을 해야 함

✅ 15.5 TEXT와 BLOB

TEXT, BLOB은 대량의 데이터를 저장할 때 사용됩니다. 두 타입은 많은 부분에서 거의 똑같은 설정이나 방식으로 작동됩니다.

단지 TEXT 타입은 문자열을 저장하는 대용량 칼럼이므로 집합, 콜레이션을 가지고 BLOB은 이진 데이터이므로 문자 집합, 콜레이션을 따지지 않는다는 차이점만 있습니다.

TEXT, BLOB 타입이 내부적으로 저장 가능한 최대 길이에 따른 4개의 타입분류

데이터 타입	필요 저장 공간 (L = 저장하고자 하는 데이터의 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, LONG VARCHAR는 `MEDIUMTEXT`의 동의어임
- 이진 데이터, 문자열을 저장하기 위한 데이터 타입은 고정길이, 가변길이 타입이 정확하게 매핑됩니다.
  (HAVETOFIND 무슨말이지..?)

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

TEXT, BLOB을 선택할때 고려할 상황

- TEXT는 오라클의 CLOB 타입과 동일한 역할을 함
  - 칼럼 하나에 저장되는 문자열이나 이진 값의 길이가 예측할 수 없이 클 때 TEXT나 BLOB을 사용한다. 하지만 다른 DBMS와는 달리 MySQL에서는 값의 크기가 4000바이트를 넘을 때 반드시 BLOB이나 TEXT를 사용해야 하는 것은 아니다. MySQL에서는 레코드의 전체 크기가 64KB를 넘지 않는 한도 내에서는 VARCHAR나 VARBINARY의 길이는 제한이 없다. 그래서 용도에 따라서는 다음 예제와 같이 4000바이트 이상의 값을 저장하는 칼럼도 VARCHAR나 VARBINARY 타입을 이용할 수 있다.
  - MySQL에서는 버전에 따라 조금씩 차이는 있지만 일반적으로 하나의 레코드는 전체 크기가 64KB를 넘어설 수 없다. VARCHAR나 VARBINARY와 같은 가변 길이 칼럼은 최대 저장 가능 크기를 포함해 64KB로 크기가 제한된다. 레코드의 전체 크기가 64KB를 넘어서서 더 큰 칼럼을 추가할 수 없다면 일부 칼럼을 TEXT나 BLOB 타입으로 전환해야 할 수도 있다.

MySQL 인덱스 레코드의 모든 칼럼의 최대 제한 크기(인덱스로 쓸 수 있는 최대 바이트)

- MyISAM은 1000B
- InnoDB with `REDUNDANT` or `COMPACT` Row Format은 767B
- '' with `DYNAMIC` or `COMPRESSED` Row Format은 3072B

자주는 아니지만 BLOB, TEXT 타입 칼럼에 인덱스 생성시 몇 바이트까지 인덱스를 생성할 것인지 명시해야 할 때도 있습니다.(최대 제한 크기 넘길 수 없음)

InnoDB에서 DYNAMIC, COMPRESSED 로우 포맷에서 문자집합에 따른 인덱스 크기

- `utf8mb4`: 최대 768글자까지만 인덱스 생성 가능
- `latin1`: 최대 3072글자까지 인덱스로 생성 가능

BLOB, TEXT의 정렬

- `max_sort_length` 시스템 변수에 설정된 길이까지만 정렬 수행함
  (기본은 1024바이트 TEXT 타입의 정렬을 더 빠르게 하려면 이 값을 줄이는게 좋음)

쿼리 특성에 따른 임시 테이블 생성
(될 수 있으면 디스크가 아닌 메모리에 생성하도록 만들기)

- `internal_tmp_mem_storage_engine` 변수 설정에 따라 MEMEORY, TempTable 스토리지 엔진 중 하나를 선택함
- 8.0부터는 MEMORY 스토리지 엔진은 TEXT, BLOB 지원을 하지 않으므로 TempTable로 설정해서 BLOB 타입이나 TEXT 타입을 포함하는 결과도 메모리를 사용할 수 있게 하는게 좋음

INSERT, UPDATE시 SQL 문장이 전송되지 못하는 현상 막기

- BLOB, TEXT를 조작하는 SQL 문장이 너무 길어져 `max_allowd_packet` 시스템 변수보다 큰 값을 갖는 SQL문장은 서버로 전송되지 못합니다.
- 따라서 대용량 BLOB, TEXT 칼럼을 사용하는 쿼리라면 위 시스템 변수를 늘려줘야 합니다.

TEXT, BLOB이 저장되는 방식을 결정하는 요소는 테이블의 ROW_FORMAT 옵션

- 별도로 지정되지 않는다면 `innodb_default_row_format` 값을 적용
- 기본 `innodb_default_row_format`은 `dynamic`으로 설정됨
- 8.0은 사용 가능한 모든 ROW_FORMAT에서는 TEXT, BLOB 칼럼의 값을 다른 레코드와 같이 저장하려 합니다.
  - 하지만, 레코드의 최대 길이 제한때문에 불가능합니다.

레코드의 최대 길이 제한: BLOB, TEXT타입 칼럼 값을 다른 레코드와 같이 저장X

- COMPACT 포맷은 나머지 포맷의 바탕이 됨
  - `DYNAMIC`은 `COMPACT` 포맷에 몇 가지 규칙이 추가된 버전
  - `COMPRESSED`는 `DYNAMIC`에 압축 관련 규칙이 추가된 버전

- COMPACT 포맷에서 저장할 수 있는 레코드 하나의 최대 길이 는 8126B

fd_blob의 길이	fd_text의 길이	fd_blob의 저장 위치	fd_text의 저장 위치
3000	3000	프라이머리 키 페이지	프라이머리 키 페이지
3000	10000	프라이머리 키 페이지	외부 페이지
10000	10000	외부 페이지	외부 페이지

  따라서 전체길이가 8126을 넘어서면 용량이 큰 칼럼 순서대로 외부 페이지로 옮겨 레코드의 크기를 8126 이하로 맞추려 합니다.

- 외부 페이지로 옮겨진 BLOB, TEXT칼럼이 16KB를 넘으면 mysql서버는 이를 다시 여러개의 외부 페이지에 저장하고 각 페이지를 체인으로 연결합니다.

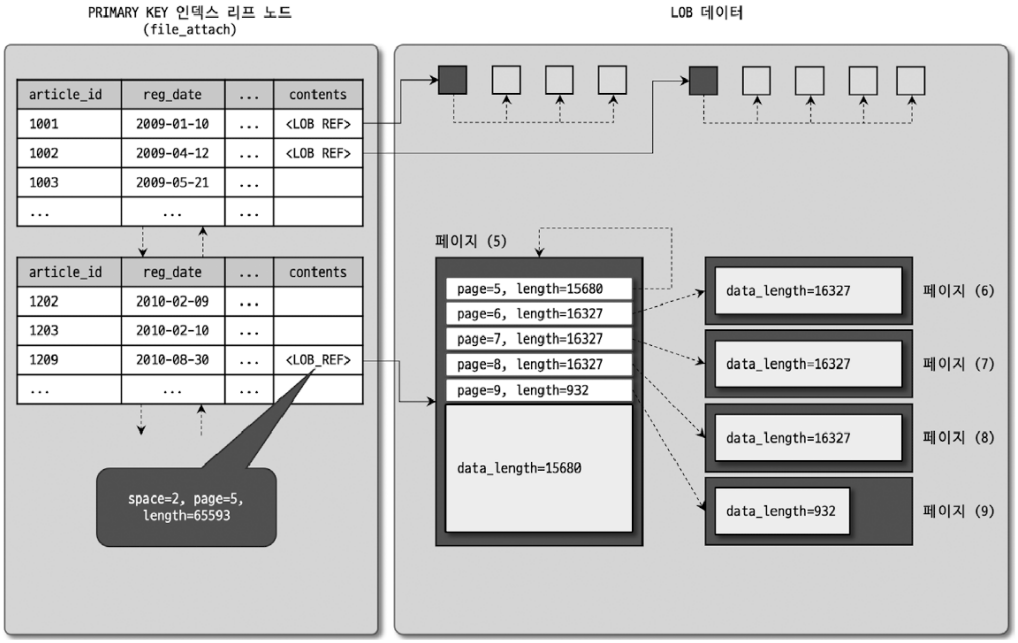

그림 15.4 BLOB이나 TEXT 칼럼의 값이 여러 개의 외부 페이지로 저장된 형태

    - 하나의 테이블에 여러 개의 BLOB, TEXT가 있다면 하나의 레코드는 여러 개의 외부 페이지 체인을 가질 수 있음

- mysql 서버는 `COMPACT`와 `REDUNDANT` 레코드 포맷을 사용하는 테이블에서 외부 페이지로 저장된 TEXT, BLOB 칼럼의 앞쪽 768바이트(BLOB prefix)만 잘라서 PK 페이지에 같이 저장합니다.
- 반면에 DYNAMIC, COMPRESSED 레코드 포맷은 BLOB prefix를 PK 페이지에 저장하지 않습니다.

- 발생할 수 있는 `Side Effect`

  → 따라서 COMPACT나 REDUNDANT 레코드 포맷의 BLOB 인덱스를 생성할 때 도움이 되기도 합니다.

  → 하지만 BLOB, TEXT 칼럼을 가진 테이블의 저장 효율을 낮추게 될 수 도 있습니다.

  → BLOB prefix는 공간을 차지하므로 PK 페이지에 저장할 수있는 레코드의 건수를 더 줄입니다. 따라서 BLOB, TEXT 칼럼을 거의 참조하지 않는 쿼리는 성능이 더 떨어집니다.

✅ 15.6 공간 데이터 타입

MySQL 서버는 OpenGIS에서 제시하는 표준 준수

- `WKT(Well Known Text)` or `WKB(Well Known Binary)`를 이용해 공간 데이터를 관리할 수 있게 지원함

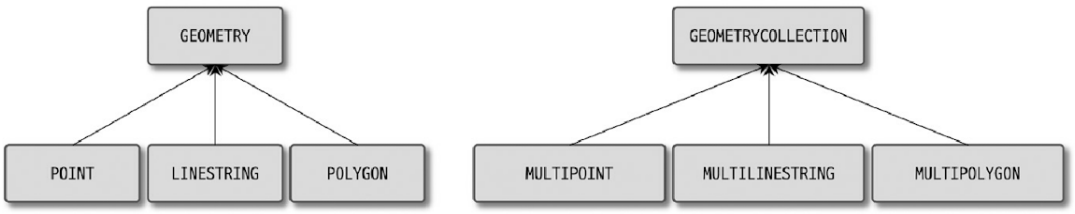

그림 15.5 공간 데이터 계층도

- `GEOMETRY` 및 해당 타입의 하위 타입은 하나의 단위 정보만 가짐
  - 단순한 위치 좌표 하나만 저장해도 되면 POINT (대부분 충분함)
  - BLOB 객체로 관리됨, 전송도 BLOB
    (BLOB을 감싸고 있는 구조 == GEOMETRY)
  - JDBC는 공간 데이터 정식 지원 X, ORM 라이브러리들은 JTS같은 라이브러리를 활용함
  - GEOMETRY 칼럼 데이터 일관적이면 mysql 공간 함수를 이용해 JDBC 지원 타입으로 변환하고 조회하는 방식도 고려해볼만함
  - 일반적인 POINT, POLYGON 데이터는 몇십 ~ 몇백 바이트 수준 이므로 성능을 크게 걱정하지 않아도 됨
- `GEOMETRYCOLLECTION` 및 해당 타입의 하위 타입은 종류별로 여러 개의 객체를 저장할 수 있습니다.

15.6.1 공간 데이터 생성

공간 데이터 타입을 mysql 서버에 생성할 때 어떤 방법으로 저장하는지에 대한 예시들

공간 데이터를 생성할 때는 아래와 같은 함수를 이용해 WKT포맷을 mysql 서버가 처리할 수 있는 이진 데이터 포맷의 데이터로 변환할 수 있습니다.
(WKT 포맷에서 보여지는 들여쓰기는 가독성을 위함임)

객체 생성 함수에서 x, y필드는 사용하는 좌표 시스템에 따라 위도 경도 혹은 단순 정숫값 등 다양한 값이 저장됨

POINT 타입

```
WKT 포맷  : POINT(x y)

객체 생성 : ST_PointFromText('POINT(x y)')
```

LINESTRING 타입

```
WKT 포맷  : LINESTRING(x0 y0,  x1 y1,  x2 y2,  x3 y3,  ...)

객체 생성 : ST_LineStringFromText('LINESTRING(x0 y0,  x1 y1,  x2 y2,  x3 y3,  ...)')
```

POLYGON 타입

```
WKT 포맷  : POLYGON((x0 y0,  x1 y1,  x2 y2,  x3 y3,  x0 y0))

객체 생성 : ST_PolygonFromText('POLYGON((x0 y0,  x1 y1,  x2 y2,  x3 y3,  x0 y0))')
```

MULTIPOINT 타입

```
WKT 포맷  : MULTIPOINT(x0 y0,  x1 y1,  x2 y2)

객체 생성 : ST_MultiPointFromText('MULTIPOINT(x0 y0,  x1 y1,  x2 y2)')
```

MULTILINESTRING 타입

```
WKT 포맷  : MULTILINESTRING((x0 y0,  x1 y1), (x2 y2,  x3 y3))

객체 생성 : ST_MultiLineStringFromText('MULTILINESTRING((x0 y0,  x1 y1), (x2 y2,  x3 y3))')
```

MULTIPOLYGON 타입

```
WKT 포맷  : 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('MULTIPOLYGON(((x0 y0,  x1 y1,  x2 y2,  x3 y3,  x0 y0)),
                                                  ((x4 y4,  x5 y5,  x6 y6,  x7 y7,  x4 y4)))')
```

GEOMETRYCOLLECTION 타입

```
WKT 포맷  : GEOMETRYCOLLECTION(POINT(x0 y0), POINT(x1 y1), LINESTRING(x2 y2,  x3 y3))

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

→ `FromText` → `FromWKB` 로 변경하면 WKT 대신 WKB를 이용

→ 위의 모든 함수들엔 두번째 인자도 SRID를 넘길 수 있음(default는 0)

15.6.2 공간 데이터 조회

아래와 같이 여러 방법 존재

1. 이진 데이터 조회(WKB or MySQL이진 포맷)
2. 텍스트 데이터 조회(WKT 포맷)
3. 공간 데이터의 속성 함수를 이용한 조회

- 첫 번째와 두 번째는 공간 데이터 타입과 관계없이 `ST_AsText()/ST_AsWKT()` 함수나 `ST_AsBinary()/ST_AsWKB()` 함수를 이용해 조회

  mysql 서버가 내부적으로 사용하는 이진 포맷의 공간 데이터 확인 가능

```
mysql> SELECT id,
              location AS internal_format, /* MySQL 서버의 내부 이진 데이터 그대로 조회 */
              ST_AsText(location) AS wkt_format,
              ST_AsBinary(location) AS wkb_format
       FROM plain_coord \G
*************************** 1. row ***************************
             id: 1
internal_format: 0x0000000001010000000000000000000000000000000000000000
     wkt_format: POINT(0 0)
     wkb_format: 0x010100000000000000000000000000000000000000
```

- 각 속성을 구분해서 조회하려면 공간 데이터 타입별로 사용할 수 있는 함수 달라짐

POINT 타입 속성 함수

```
mysql> SET @poi:=ST_PointFromText('POINT(37.544738 127.039074)', 4326);

mysql> SELECT
         ST_SRID(@poi) AS srid,
         ST_X(@poi) AS coord_x,
         ST_Y(@poi) AS coord_y,
         ST_Latitude(@poi) AS coord_latitude,
         ST_Longitude(@poi) AS coord_longitude;
+------+-----------+-------------------+----------------+--------------------+
| srid | coord_x   | coord_y           | coord_latitude | coord_longitude    |
+------+-----------+-------------------+----------------+--------------------+
| 4326 | 37.544738 | 127.03907400000001 |      37.544738 | 127.03907400000001 |
+------+-----------+-------------------+----------------+--------------------+
```

LINESTRING과 MULTILINESTRING 타입 속성 함수

```
mysql> SET @line := ST_LineStringFromText('LINESTRING(37.55601011174991 127.03600689589169,
                                                      37.55601011174991 127.05866710410828,
                                                      37.53804388825009 127.05866710410828,
                                                      37.53804388825009 127.03600689589169)')

mysql> SELECT
         ST_AsText(ST_StartPoint(@line)),
         ST_AsText(ST_EndPoint(@line)),
         ST_AsText(ST_PointN(@line, 2)),
         ST_IsClosed(@line),
         ST_Length(@line),
         ST_NumPoints(@line) \G

*************************** 1. row ***************************
ST_AsText(ST_StartPoint(@line)): POINT(37.55601011174991 127.03600689589169)
  ST_AsText(ST_EndPoint(@line)): POINT(37.53804388825009 127.03600689589169)
 ST_AsText(ST_PointN(@line, 2)): POINT(37.55601011174991 127.05866710410828)
             ST_IsClosed(@line): 0
               ST_Length(@line): 0.0632866399330112
            ST_NumPoints(@line): 4
```

POLYGON과 MULTIPOLYGON 속성 함수

```
mysql> SET @polygon := ST_PolygonFromText('POLYGON((37.55601011174991 127.03600689589169,
                                                    37.55601011174991 127.05866710410828,
                                                    37.53804388825009 127.05866710410828,
                                                    37.53804388825009 127.03600689589169,
                                                    37.55601011174991 127.03600689589169))',
                                           4326);

mysql> SELECT
         ST_Area(@polygon),
         ST_AsText(ST_ExteriorRing(@polygon)),
         ST_AsText(ST_InteriorRingN(@polygon, 1)),
         ST_NumInteriorRing(@polygon),
         ST_NumInteriorRings(@polygon);
*************************** 1. row ***************************
                      ST_Area(@polygon): 3993026.2901834054
    ST_AsText(ST_ExteriorRing(@polygon)): LINESTRING(37.55601011174991 127.03600689589169,
                                                     37.55601011174991 127.05866710410828,
                                                     37.53804388825009 127.05866710410828,
                                                     37.53804388825009 127.03600689589169,
                                                     37.55601011174991 127.03600689589169)
ST_AsText(ST_InteriorRingN(@polygon, 1)): NULL
             ST_NumInteriorRing(@polygon): 0
            ST_NumInteriorRings(@polygon): 0
```

✅ 15.7 JSON 타입

- JSON 타입은 mysql 5.7부터 지원을 시작함
- 8.0으로 올라오며 기능 추가 및 개선이 이루어짐
- BLOB에도 저장할 수 있지만, JSON 칼럼은 문자열이 아닌 MongoDB와 같이 바이너리 포맷(BSON) 으로 변환하여 저장합니다.

15.7.1 저장 방식

- JSON 칼럼에 저장되는 값을 BSON(바이너리 포맷)으로 변환하여 저장하므로 BLOB, TEXT보다 공간 효율이 높습니다.

JSON 칼럼의 값이 이진 포맷으로 변환됐을때 바이트 확인 예제

```
# JSON 칼럼의 값이 이진 포맷으로 변환됐을때 길이가 몇 바이트인지 확인
CREATE TABLE tb_json
(
    id INT,
    fd JSON
);

# user_id 필드 값을 정수 타입으로 저장
INSERT INTO tb_json
VALUES (1, '{
  "user_id": 1234567890
}'),
       (2, '{
         "user_id": "1234567890"
       }');

# user_id 필드의 값을 문자열 저장
SELECT id, fd, JSON_TYPE(fd -> "$.user_id") AS field_type, JS
FROM tb_json;
```

	id	fd	field_type	byte_size
1	1	{"user_id": 123456789	INTEGER	23
2	2	{"user_id": "12345678	STRING	30

- JSON 값을 이진 포맷으로 변환하면 7바이트의 공간차이가 발생합니다.

JSON 도큐먼트가 이진 데이터로 변환되어 저장되는 방식

```
-- // JSON 도큐먼트 { "a": "x", "b": "y", "c": "z" }
mysql> SELECT JSON_STORAGE_SIZE('{ "a": "x", "b": "y", "c": "z" }') AS binary_length;
+---------------+
| binary_length |
+---------------+
|            35 |
+---------------+
```

```
# 35 바이트의 이진 데이터
00 03 00 22 00 19 00 01 00 1A
00 01 00 1B 00 01 00 0C 1C 00
0C 1E 00 0C 20 00 61 62 63 01
78 01 79 01 7A
```

이진 데이터는 아래와 같이 24개의 필드로 구성되어 있습니다. 각 필드는 값 특성에 맞게 1개 이상의 바이트를 차지합니다.

표 15.1 이진 포맷 JSON 데이터 필드

필드 순서	바이트 수	주소(Offset)	이진값		문자열	십진 숫자값	설명
1	1		00			0	type(JSONB_TYPE_SMALL_OBJECT)
2	2	0	03	00		3	JSON 어트리뷰트 개수
3	2	2	22	00		34	JSON 도큐먼트 길이(바이트 수)
4	2	4	19	00		25	첫 번째 키 주소(Offset)
5	2	6	01	00		1	첫 번째 키 길이(바이트 수)
6	2	8	1A	00		26	두 번째 키 주소(Offset)
7	2	10	01	00		1	두 번째 키 길이(바이트 수)
8	2	12	1B	00		27	세 번째 키 주소(Offset)
9	2	14	01	00		1	세 번째 키 길이(바이트 수)
10	1	16	0C			12	첫 번째 값 타입(JSONB_TYPE_STRING)
11	2	17	1C	00		28	첫 번째 값 주소(Offset)
12	1	19	0C			12	두 번째 값 타입(JSONB_TYPE_STRING)
13	2	20	1E	00		30	두 번째 값 주소(Offset)
14	1	22	0C			12	세 번째 값 타입(JSONB_TYPE_STRING)
15	2	23	20	00		32	세 번째 값 주소(Offset)
16	1	25	61		a		첫 번째 키
17	1	26	62		b		두 번째 키
18	1	27	63		c		세 번째 키
19	1	28	01			1	첫 번째 값 길이(바이트 수)
20	1	29	78		x		첫 번째 값
21	1	30	01			1	두 번째 값 길이(바이트 수)
22	1	31	79		y		두 번째 값
23	1	32	01			1	세 번째 값 길이(바이트 수)
24	1	33	7A		z		세 번째 값

- `이진값` : 실제 바이너리 필드 값
- `주소(Offset)` : 첫번째 1바이트를 제외하고 오프셋을 의미합니다.
- JSON 도큐먼트를 구성하는 모든 키의 위치와 키의 이름이 각 JSON 필드의 값 보다 먼저 나열돼 있습니다.
    - 따라서 JSON 칼럼의 특정 필드 참조 혹은 값만 업데이트(길이가 변경되지 않는 부분 업데이트)의 경우 JSON 칼럼의 값을 모두 읽지 않아도 즉시 원하는 필드를 수정할 수 있습니다.

- 매우 큰 용량의 JSON 도큐먼트는 16KB단위로 여러 개의 데이터 페이지로 나눠 저장됨
  - 5.7까지 BLOB 데이터(JSON은 내부적으로 BLOB 타입을 사용)는 여러 개의 BLOB 페이지를 단순 링크드 리스트 처럼 관리함
  - **8.0부터는 BLOB 페이지들의 인덱스를 관리하고 각 인덱스는 실제 BLOB 데이터를 가진 페이지들의 링크를 갖도록 개선됨**
- 개선된 BLOB 페이지에서 JSON 필드의 부분 업데이트를 할때, mysql 서버는 BLOB 페이지 인덱스와 JSON 칼럼의 각 필드 주소 정보를 이용해 필요한 부분만 업데이트 함

15.7.2 부분 업데이트 성능

- JSON 칼럼의 부분 업데이트 기능은 JSON_SET(), JSON_REPLACE(), JSON_REMOVE() 함수를 이용해 **JSON 도큐먼트의 특정 필드 값을 변경, 삭제하는 경우에만 작동합니다.**

```
# 1234567890 -> 12345로 변경
UPDATE tb_json
SET fd=JSON_SET(fd, '$.user_id', "12345")
WHERE id = 2;

SELECT id, fd, JSON_STORAGE_SIZE(fd), JSON_STORAGE_FREE(fd)
FROM tb_json;
```

	id	fd	`JSON_STORAGE...	`JSON_STORAGE...
1	1	{"user_id": 123456789	23	0
2	2	{"user_id": "12345"}	30	5

- 1234567890 → 12345로 변경했기 때문에 10바이트 중에서 5바이트는 비웠기에 JSON_STORAGE_FREE의 값이 5바이트가 표시됩니다.
- 반면에 JSON_STORAGE_SIZE의 크기는 그대로 입니다. 따라서 디스크 저장공간은 그대로지만 실제 공간은 5바이트가 비워졌음을 의미합니다.
- 이를 통해 필드 값 변경 작업이 `부분 업데이트`로 처리됐는지 유추할 수 있습니다.

```
# 10바이트를 초과하도록 변경
UPDATE tb_json
SET fd=JSON_SET(fd, '$.user_id', "12345678901")
WHERE id = 2;

SELECT id, fd, JSON_STORAGE_SIZE(fd), JSON_STORAGE_FREE(fd)
FROM tb_json;
```

	id	fd	`JSON_STORAGE...	`JSON_STORAGE...
1	1	{"user_id": 123456789	23	0
2	2	{"user_id": "12345678901"}	31	0

- 기존 공간 10바이트보다 더 큰 11바이트로 변경했기 때문에 부분 업데이트 방식으로 처리되지 못했습니다.
  - JSON_STORAGE_FREE도 0으로 초기화

특정 조건에서 매우 빠른 업데이트 성능을 보여주는 부분 업데이트 기능

- JSON 칼럼의 값은 내부적으로 LONGBLOB 타입으로 저장되고 최대 4GB의 값을 가질 수 있습니다.
- 1MB만 저장돼도 16KB 페이지 64개를 사용합니다. 만약 부분 업데이트를 할 수 없다면 64개의 데이터 페이지 전체를 다시 디스크로 기록합니다.

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

INSERT INTO tb_json(id, fd)
VALUES (1, JSON_OBJECT('name', 'Matt', 'visits', 0, 'data', R
       (2, JSON_OBJECT('name', 'Matt', 'visits', 0, 'data', R
       (3, JSON_OBJECT('name', 'Matt', 'visits', 0, 'data', R
       (4, JSON_OBJECT('name', 'Matt', 'visits', 0, 'data', R
```

```
INSERT INTO tb_json(id, fd)
SELECT id + 5, fd
FROM tb_json;

INSERT INTO tb_json(id, fd)
SELECT id + 10, fd
FROM tb_json;

# 부분 업데이트를 사용할 수 없는 경우
UPDATE tb_json
SET fd = JSON_SET(fd, '$.name', "Matt Lee")

16 rows affected in 1 s 202 ms

# 부분 업데이트를 사용하는 경우
UPDATE tb_json
SET fd = JSON_SET(fd, '$.name', "Kit")

16 rows affected in 548 ms
```

- 대략적으로 2배 이상의 차이가 발생함을 알 수 있습니다.

- mysql 서버는 복제를 사용하기 때문에 JSON 변경 내용을 바이너리 로그에 기록해야 합니다.
- 이때 JSON의 데이터를 모두 기록하게 되는데 설정의 변경으로 변경된 내용들만 바이너리 로그에 기록되게 하면 성능을 훨씬 빠르게 만들 수 있습니다.

```
# 변경되는 JSON 데이터만 바이너리 로그에 기록하도록 설정하고 실험
SET binlog_format = ROW;
SET binlog_row_value_options = PARTIAL_JSON;
SET binlog_row_image = MINIMAL;

UPDATE tb_json
SET fd = JSON_SET(fd, '$.name', "Matt Lee")

16 rows affected in 952 ms
```

```
UPDATE tb_json
SET fd = JSON_SET(fd, '$.name', "Kit")

16 rows affected in 150 ms
```

- 추가로 바이너리 로그의 포맷을 STATEMENT로 변경해도 거의 동일한 성능 향상 효과를 얻을 수 있습니다.

> **주의** `JSON` 칼럼의 부분 업데이트 최적화 효과를 얻기 위해서는 `binlog_row_value_options` 시스템 변수와 `binlog_row_image` 시스템 변수의 변경도 필요하지만 `JSON` 칼럼을 가진 테이블의 프라이머리 키가 필수적이다. 테이블의 프라이머리 키가 없다면 MySQL 복제에서 레플리카 서버는 업데이트할 레코드를 식별하기 위서 레코드의 모든 칼럼을 필요로 한다. 그래서 테이블의 프라이머리 키가 없는 경우, 위의 두 시스템 변수와 관계없이 `JSON` 칼럼을 포함해서 레코드의 모든 칼럼을 바이너리 로그에 기록해야 하므로 부분 업데이트의 성능은 느려진다.

한계점

다만 단순히 정수 변경이 아니라 문자열 경우 저장되는 길이에 따라 부분 업데이트가 안될 수 도 있습니다.

따라서 미리 필드가 가질 수 있는 최대 길이의 값으로 초기화하거나 애플리케이션에서 추가로 padding하여 고정 길이 문자열을 만들어서 저장하는 방법으로 부분 업데이트 기능을 활용할 수 있습니다.

15.7.3 JSON 타입 콜레이션과 비교

JSON 칼럼에 저장되는 데이터, JSON 칼럼으로부터 가공되어 나온 결과물은 모두 `utf8mb4 문자 집합 + utf8mb4_bin 콜레이션`입니다.

`바이너리 콜레이션`이므로 대소문자 구분은 물론 액센트 문자 등도 구분하여 비교합니다.

```
mysql> SET @user1 = JSON_OBJECT('name', 'Matt');
mysql> SELECT CHARSET(@user1), COLLATION(@user1);
+-----------------+-------------------+
| CHARSET(@user1) | COLLATION(@user1) |
+-----------------+-------------------+
| utf8mb4         | utf8mb4_bin       |
+-----------------+-------------------+

mysql> SET @user2 = JSON_OBJECT('name', 'matt');
mysql> SELECT @user1=@user2;
+---------------+
| @user1=@user2 |
+---------------+
|             0 | ==> FALSE
+---------------+
```

→ 대소문자가 서로 다름을 알 수 있음

15.7.4 JSON 칼럼 선택

- BLOB, TEXT와 달리 JSON 타입은 이진 포맷 컴팩션 저장, 필요한 경우 부분 업데이트를 통한 빠른 변경 등 다양한 기능을 제공합니다. 따라서 JSON 데이터를 저장한다면 JSON 칼럼을 선택하는게 좋습니다.

JSON 칼럼 구성 vs 정규화된 칼럼 구성 테이블

JSON 칼럼만으로 구성된 테이블

```
mysql> CREATE TABLE tb_json (
         doc JSON NOT NULL,
         id BIGINT AS (doc->>'$.id') STORED NOT NULL,
         PRIMARY KEY (id)
       );

mysql> INSERT INTO tb_json (doc) VALUES
         ('{"id":1, "name":"Matt"}'),
         ('{"id":2, "name":"Esther"}');
```

정규화된 칼럼만으로 구성된 테이블

```
mysql> CREATE TABLE tb_column (
         id BIGINT NOT NULL,
         name VARCHAR(50) NOT NULL,
         PRIMARY KEY(id)
       );

mysql> INSERT INTO tb_column VALUES
         (1, 'Matt'),
         (2, 'Esther');
```

- 성능으로만 따지면 정규화된 칼럼이 JSON 칼럼 보다는 이점을 갖습니다.
  - JSON 칼럼은 각 필드의 이름이 데이터 파일에 매번 저장돼야 함
  - 레코드 건수가 많아질수록 필드 이름이 차지하는 공간이 커짐

- JSON 칼럼을 압축을 한다면?
  - 압축된 페이지, 해제된 페이지의 공존으로 메모리, CPU 효율을 모두 떨어트림

- 정규화된 칼럼을 사용하면?
  - BLOB, TEXT와 같이 대용량 데이터의 경우 외부 페이지로 관리됩니다. 응용 프로그램 요건에 맞게 적절히 응용하면 메모리 효율, 쿼리 성능을 더 끌어올립니다.
  - 반면에 JSON은 선별적으로 접근해야 하므로 성능 향상은 기대하기 어렵습니다.

- JSON 칼럼의 장점?
  - 각 레코드가 가지는 속성들이 너무 상이하고 다양하고 레코드별로 선택적으로 값을 가지고 있는 경우
  - 단, 가능하면 중요도가 낮은 것일수록 좋습니다.
    (그러면 검색 조건이나 쿼리에서 자주 접근할 가능성이 낮으므로)
  - 정규화된 테이블 구조를 유지하면 너무 테이블의 개수가 많아질 수 도 있으므로 중요도에 따라 JSON 칼럼에 비정규화된 형태로 저장할 수 있음

✅ 15.8 가상 칼럼(파생 칼럼)

일반적인 DBMS에선 Virtual Column이라 하지만, mysql 서버는 Generated Column이라는 이름으로 소개됩니다.

저장되는 데이터의 종류를 한정하는 데이터 타입은 아님

- 가상 칼럼의 구분
  1. 가상 칼럼(Virtual Column)
  2. 스토어드 칼럼(Stored Column)

가상 칼럼, 스토어드 칼럼 사용한 테이블 생성 예제

```
# 가상 칼럼(Virtual Column) 사용 예제
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)
);

# 스토어드 칼럼(Stored olumn) 사용 예제
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)
);
```

- AS 절로 계산식을 정의함
- VIRTUAL, STORED와 키워드 정의가 없으면 default로 VIRTUAL로 칼럼을 생성함
- 가상 칼럼은 다른 칼럼의 값을 참조해 계산된 값을 관리하므로 항상 AS 절 뒤에는 계산식이나 데이터 가공을 위한 표현식을 정의합니다.
  - 가상 칼럼의 표현식은 입력이 동일하면 시점과 관계없이 결과가 항상 동일한 (DETERMINISTIC) 표현식만 사용 가능
  - 사용자 변수, NOT-DETERMINISTIC 옵션의 함수나 표현식 사용 불가
  - 8.0버전까지는 서브쿼리나, 스토어드 프로그램을 사용할 수는 없습니다.

- 가상 칼럼, 스토어드 칼럼 모두 다른 칼럼의 값을 참조해 새로운 값을 만들어 관리하는 공통점이 존재합니다.
- 차이점

  - 가상 칼럼(Virtual Column)
    - 칼럼의 값이 디스크에 저장되지 않음
    - 칼럼의 구조 변경은 테이블 리빌드를 필요로 하지 않음
    - 칼럼의 값은 레코드가 읽히기 전 또는 BEFORE 트리거 실행 직후에 계산되어 만들어짐

  - 스토어드 칼럼(Stored Column)
    - 칼럼의 값이 물리적으로 디스크에 저장됨
    - 칼럼의 구조 변경은 다른 일반 테이블과 같이 필요 시 테이블 리빌드 방식으로 처리됨
    - INSERT와 UPDATE 시점에만 칼럼의 값이 계산됨

  - 가장 큰 차이점은 계산된 칼럼의 값이 실제 디스크에 저장되는지 여부입니다.

    - 나머지 차이점은 디스크 저장 여부로 인한 결과임

- 다만, 가상 칼럼은 무조건 디스크에 저장되지 않는것은 아닙니다.
    - 가상 칼럼에 인덱스를 생성하면, 테이블 레코드는 가상 칼럼을 포함하지 않아도 인덱스는 계산된 값을 저장합니다.
    - 따라서 인덱스가 생상된 가상 칼럼의 경우 필요하다면 인덱스의 리빌드 작업이 필요합니다.

- 8.0부터 도입된 함수기반 인덱스는 가상 칼럼에 인덱스를 생성하는 방식으로 작동함
    - 테이블 조회시 가상 칼럼이 결과에 표시되진 않는다는 차이점이 있지만, 내부적으로 가상 칼럼과 동일하게 처리함

- 가상 칼럼 vs 스토어드 칼럼
    - 가상 칼럼은 조회시점마다 계산하므로 계산이 복잡하고 오래걸린다면 스토어드 칼럼으로 변경하는게 성능 향상에 도움이 됩니다.
    - 계산 속도는 빠르고 계산 결과가 많은 공간을 차지하면 가상 칼럼이 저장 공간, 메모리 효율면에서 이득
- 결론
    - `CPU 사용량을 조금 높여서 디스크 부하를 낮출 것이냐(가상 칼럼)` VS `디스크 사용량을 조금 높여서 CPU 사용량을 낮출것이냐(스토어드 칼럼)`