

week1

▼ 11.1 쿼리 작성과 연관된 시스템 변수

MySQL 서버의 시스템 설정에 따라 SQL 작성 규칙이 달라질 수 있다.

1. sql 모드

- STRICT_ALL_TABLES & STRICT_TRANS_TABLES
- ANSI_QUOTES
- ONLY_FULL_GROUP_BY
- PIPE_AS_CONCAT
- PAD_CHAR_TO_FULL_LENGTH
- NO_BACKSLASH_ESCAPES
- IGNORE_SPACE
- REAL_AS_FLOAT
- NO_ZERO_IN_DATE & NO_ZERO_DATE
- ANSI
- TRADITIONAL

2. 대소문자 구분

유닉스 계열의 OS에서는 대소문자 구분을 하지만, 윈도우에서는 구분하지 않음. 이는 MySQL 데이터를 리눅스로 가져오거나 가져갈 때 문제가 될 수 있는데, 이를 방지하기 위해 `lower_case_table_names` 시스템 변수를 설정할 수 있다.

0: 기본값 (대소문자 구분 O)

1: 모두 소문자 (대소문자 구분 X)

2: 윈도우와 macOS에서 가능하며, 저장은 대소문자를 구분하지만 쿼리에서는 구분하지 않음

기본적으로는 대문자, 혹은 소문자만 쓰는 것이 호환성에 좋다.

3. MySQL 예약어

DB나 table, column 등의 이름을 예약어와 동일하게 생성하려면 항상 역따옴표(`)

로 감싸야한다. (그러나 이는 개발이나 유지보수 측면에서 매우 성가신 일이므로 권장되지 않는다) (가령, int라는 테이블을 만들고 싶다면

`int` 라고 쳐야함)

가능한 예약어와 같은 이름은 피하자

▼ 11.2 메뉴얼의 SQL 문법 표기를 읽는 법

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
      [INTO] tbl_name
      [PARTITION (partition_name [, partition_name] ...)]
      [(col_name [, col_name] ...)]
      { {VALUES | VALUE} (value_list) [, (value_list)] ... }
      [AS row_alias[(col_alias [, col_alias] ...)]]
      [ON DUPLICATE KEY UPDATE assignment_list]
```

- 대괄호(`[]`): 해당 키워드나 표현식 자체가 선택 사항임
- 중괄호(`{ }`): 괄호 내의 아이템 중에서 반드시 하나를 사용해야 함
- 이텔릭체: 사용자가 선택해서 작성하는 토큰
- 파이프(`|`): 키워드나 표현식 중에서 하나만 선택해서 사용해야 함
- 줄임표(`...`): 표현식의 조합이 반복될 수 있음

▼ 11.3 MySQL 연산자와 내장 함수

(1) 문자열

- SQL 표준에서 문자열은 홑따옴표(`' '`)를 사용한다.
- MySQL에서 문자열은 쌍따옴표를 사용할 수도 있다.

```
-- ANSI 표준
SELECT * FROM departments WHERE dept_no = 'd001';
SELECT * FROM departments WHERE dept_no = 'd'001';
SELECT * FROM departments WHERE dept_no = 'd'"001';

-- MySQL에서만 사용 가능
```

```
SELECT * FROM departments WHERE dept_no = "d'"001";
SELECT * FROM departments WHERE dept_no = "d""001";
```

(2) 숫자

- 숫자는 따옴표 없이 그냥 입력하면 된다.
- 숫자를 문자열 형태로 따옴표를 사용하더라도 비교 대상이 숫자 값이거나 숫자 타입의 칼럼이면 MySQL 서버에서 자동 변환된다.
- 하지만 숫자가 저장된 문자열 칼럼에서 숫자 값을 상수로 비교하면 칼럼의 모든 문자열 값을 숫자로 변환해서 비교를 수행해야 하므로 주의해야 한다.

```
-- rows 칼럼: 1 건
EXPLAIN SELECT * FROM employees WHERE emp_no = '10001';
+----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | employees  | NULL       | const | PRIMARY      | PRIMARY | 4 | const | 1 | 100.00 | NULL |
```

```
CREATE TABLE IF NOT EXISTS str_employees SELECT * FROM employees;
ALTER TABLE str_employees MODIFY emp_no varchar(10);
```

```
-- rows 칼럼: 299198 건
EXPLAIN SELECT * FROM str_employees WHERE emp_no = '10001';
+----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
```

```

---+-----+
| id | select_type | table          | partitions | type |
possible_keys | key  | key_len | ref  | rows  | filtere
d | Extra      |
+---+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+
---+-----+
| 1 | SIMPLE      | str_employees | NULL      | ALL  |
NULL      | NULL | NULL    | NULL    | 299198 | 10.0
0 | Using where |
+---+-----+-----+-----+-----+
-----+-----+-----+-----+-----+
---+-----+

```

(3) 날짜

- MySQL에서는 정해진 형태의 날짜 포맷으로 표기하면 MySQL 서버가 자동으로 `DATE` 나 `DATETIME` 값으로 변환하기 때문에 `STR_TO_DATE()` 같은 함수를 사용하지 않아도 된다.

(4) 불리언

- MySQL에서 `BOOL` 이나 `BOOLEAN` 타입은 사실 `TINYINT` 타입이다.
- 꼭 불리언 타입으로 사용해야 한다면 `ENUM` 타입으로 관리하는 것이 실수를 줄이고 명확하게 사용하는 방법이다.

```

-- 테스트용 테이블 생성
CREATE TABLE tb_boolean (bool_value BOOLEAN);
INSERT INTO tb_boolean VALUES (FALSE);

-- 데이터 추가
INSERT INTO tb_boolean VALUES (FALSE), (TRUE), (2), (3),
(4), (5);

-- BOOLEAN 타입이 TINYINT로 저장됨
SELECT * FROM tb_boolean WHERE bool_value IN (FALSE, TRUE);
+-----+
| bool_value |

```

```

+-----+
|          0 |
|          1 |
+-----+

```

3.2) MySQL 연산자

(1) 동등 비교

- 동등 비교 연산자는 `=` 와 `<=>` (NULL-Safe)가 있다.

```
SELECT 1 = 1, NULL = NULL, 1 = NULL;
```

```

+-----+-----+-----+
| 1 = 1 | NULL = NULL | 1 = NULL |
+-----+-----+-----+
|      1 |          NULL |        NULL |
+-----+-----+-----+

```

```
SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL;
```

```

+-----+-----+-----+
| 1 <=> 1 | NULL <=> NULL | 1 <=> NULL |
+-----+-----+-----+
|      1 |          1 |          0 |
+-----+-----+-----+

```

(2) 부정 비교

- 부정 비교 연산자는 `<>` 와 `!=` 가 있다.

(3) NOT 연산자

- 연산의 결과를 반대로(부정) 만드는 연산자로 `NOT` 또는 `!` 를 사용한다.

```
SELECT !1, !FALSE, NOT 1, NOT 0, NOT (1=1);
```

```

+-----+-----+-----+-----+-----+
| !1 | !FALSE | NOT 1 | NOT 0 | NOT (1=1) |
+-----+-----+-----+-----+-----+
|  0 |      1 |    0 |    1 |          0 |
+-----+-----+-----+-----+-----+

```

(4) AND와 OR 연산자

- **AND** 연산은 **&&** **OR** 연산은 **||** 를 사용한다.
- 오라클에서는 **OR** 연산자(**||**)는 문자열을 결합하는 연산자로 사용된다.
 - 오라클에서 쓰는 방식으로 변경하려면 **sql_mode** 에서 **PIPE_AS_CONCAT** 을 설정하면 된다.
- **AND**, **OR** 연산자 우선순위
 - 순서와 관계없이 **AND** 연산이 **OR** 연산보다 우선 처리된다. 괄호로 묶은 경우에는 괄호가 우선 처리된다.

(5) 나누기와 나머지 연산자

- 나누기 연산은 **/** 또는 **DIV** 연산자를 사용한다.
- 나머지를 가져오는 연산자로는 **%** 또는 **MOD** 를 사용한다.

```
SELECT 29 / 9,
        29 DIV 9,
        29 % 9,
        MOD(29, 9),
        29 MOD 9;
```

29 / 9	29 DIV 9	29 % 9	MOD(29, 9)	29 MOD 9
3.2222	3	2	2	2

(6) REGEXP 연산자

- 문자열 값이 어떤 패턴을 만족하는지 확인하는 연산자로, **REGEXP** 와 **RLIKE** 가 있다.
 - **RLIKE** 는 오른쪽 일치 연산자가 아닌 정규 표현식 연산자이다.
 - **REGEXP** 조건의 비교는 인덱스 레인지 스캔을 사용할 수 없다. → 가능하면 데이터 조회 범위를 줄일 수 있는 조건과 함께 **REGEXP** 연산자를 사용하길 권장 (정규 표현식 관련 내용은 문서 참조)

```
SELECT 'abc' REGEXP '^[x-z]',
        'abc' RLIKE '^[x-z]';
```

'abc' REGEXP '^[x-z]'	'abc' RLIKE '^[x-z]'
+-----+	
0	0
+-----+	

(7) LIKE 연산자

- **REGEXP** 보다는 단순한 문자열 패턴 비교 연산자 (사용 빈도는 **LIKE** 가 더 높다)
- 와일드카드 문자(**%**, **_**)를 사용하여 단순한 문자열 패턴을 비교하는 연산자이다.
 - **%** : 0개 또는 1개 이상의 문자에 일치
 - **_** : 1개의 문자에 일치
- **LIKE** 연산자를 사용할 때 와일드카드가 검색어의 뒤쪽에 있다면 인덱스를 이용해 처리할 수 있지만, 앞쪽에 있다면 인덱스를 사용할 수 없다.

-- '%' 연산자를 사용한 LIKE

```
SELECT COUNT(*)
FROM employees
WHERE first_name LIKE 'Christ%';
```

-- '%' 연산자가 선행일치로 되어 있으면 인덱스를 사용하지 못함

```
SELECT COUNT(*)
FROM employees
WHERE first_name LIKE '%rist';
```

(8) BETWEEN 연산자

- **BETWEEN** 연산자는 "크거나 같다"와 "작거나 같다"라는 두 개의 연산자를 하나로 합친 연산자이다.
- **BETWEEN** 은 조건에 해당하는 인덱스의 모든 범위를 검색하기 때문에 범위를 줄이는 것이 중요하다. → **IN** 연산자를 사용해 **BETWEEN** 의 범위를 줄일 수 있다. → **IN (subquery)** 또는 단순히 조인으로 쿼리를 작성할 수도 있다. (MySQL 8.0 버전부터 세미조인을 지원한다.)

```
-- BETWEEN 연산자의 범위를 줄이지 않은 경우 (rows = 149739)
SELECT * FROM dept_emp USE INDEX(PRIMARY)
WHERE dept_no BETWEEN 'd003' AND 'd005' AND emp_no = '10001';

-- BETWEEN 연산자의 범위를 IN (subquery)로 줄인 경우 (rows = 3)
SELECT * FROM dept_emp USE INDEX(PRIMARY)
WHERE dept_no IN ('d003', 'd004', 'd005') AND emp_no = '10001';
```

(9) IN 연산자

- 여러개의 값에 대해 **동등 비교** 연산을 수행하는 연산자이다.
 - 상수가 사용된 경우 : `IN (?, ?, ?)`
 - 서브쿼리가 사용된 경우 : `IN (SELECT .. FROM ..)`
- 여러 개의 값이 비교되지만, 범위로 검색하는 것이 아닌 여러 번의 동등 비교로 실행하기 때문에 일반적으로 빠르게 처리된다.
- 하지만, `NOT IN`의 경우에는 부정형 비교이기 때문에 인덱스로 처리 범위를 줄일 수 없게 되어 인덱스 풀스캔이 된다.
- MySQL 8.0 버전 ~
 - `IN (subquery)`의 세미 조인 최적화가 많이 안정되었다.
 - `IN` 절에 튜플을 나열해도 인덱스를 최적으로 사용할 수 있게 개선되었다.

3.3) MySQL 내장 함수

기본적인 기능의 SQL 함수는 대부분 제공되지만, DBMS 별로 호환되지는 않는다.

(1) NULL 값 비교 대체

- `IFNULL()` : `NULL` 일 경우 대체할 값이나 칼럼을 설정
- `ISNULL()` : `NULL` 인지 아닌지 비교

(2) 현재 시각 조회

- `NOW()` : 쿼리가 실행된 시점의 시간 (하나의 SQL문에서 같은 값을 공유)
- `SYSDATE()` : 호출되는 시점에 따라 결과값이 달라짐 (사용에 주의)

- `SYSDATE()` 의 잠재적인 문제
 - `SYSDATE()` 함수가 사용된 SQL은 레플리카 서버에서 안정적으로 복제되지 못한다.
 - `SYSDATE()` 함수와 비교되는 칼럼은 인덱스를 효율적으로 사용하지 못한다.

(3) 날짜와 시간 포맷

- 시간은 `%Y-%m-%d %H:%i:%s` 와 같은 지정자를 사용하여 적절하게 변환한다.
 - `DATE_FORMAT()`
 - `STR_TO_DATE()`

```
SELECT DATE_FORMAT(NOW(), '%Y/%m/%d');
```

(4) 날짜와 시간 연산

- `DATE_ADD()` : 날짜 더하기
- `DATE_SUB()` : 날짜 빼기

```
-- 날짜 더하기
SELECT DATE_ADD(NOW(), INTERVAL 1 DAY) AS tomorrow;
SELECT DATE_ADD(NOW(), INTERVAL -1 DAY) AS yesterday;

-- 날짜 빼기
SELECT DATE_SUB(NOW(), INTERVAL 1 DAY) AS yesterday;
```

→ 사실 `DATE_ADD()` 로 더하거나 빼는 처리를 모두 할 수 있기 때문에 `DATE_SUB()` 는 크게 필요하지 않다.

(5) 타임스탬프 연산

- `1970-01-01 00:00:00` 으로부터 경과된 초의 수를 반환한다. → [\[참고\] 유닉스 시간](#)
 - `UNIX_TIMESTAMP()`
 - `FROM_UNIXTIME()`

(6) 문자열 처리

- 문자를 덧붙여서 지정된 길이의 문자열로 만든다.

- RPAD()
- LPAD()

```
-- 문자 길이보다 짧은 경우 잘림
SELECT RPAD('JAYSON', 3, '-');
+-----+
| RPAD('JAYSON', 3, '-') |
+-----+
| JAY                      |
+-----+

SELECT LPAD('JAYSON', 5, '-');
+-----+
| LPAD('JAYSON', 5, '-') |
+-----+
| JAYSO                    |
+-----+

-- 문자 길이보다 긴 경우 지정된 문자를 덧붙임
SELECT RPAD('JAYSON', 7, '-');
+-----+
| RPAD('JAYSON', 7, '-') |
+-----+
| JAYSON-                 |
+-----+

SELECT LPAD('JAYSON', 7, '-');
+-----+
| LPAD('JAYSON', 7, '-') |
+-----+
| -JAYSON                 |
+-----+
```

- 연속된 공백 문자를 제거한다.

- RTRIM()
- LTRIM()
- TRIM()

(7) 문자열 결합

- 여러 개의 문자열을 연결해서 하나의 문자열로 반환한다.

- `CONCAT()`
- `CONCAT_WS()`

(8) GROUP BY 문자열 결합

- `GROUP_CONCAT()`
 - 주로 `GROUP BY` 와 함께 사용하며, 여러 레코드의 값을 병합해서 하나의 값을 만들어내는 그룹 함수이다.

```
-- 먼저 값을 정렬한 뒤 연결함
SELECT GROUP_CONCAT(DISTINCT dept_no ORDER BY emp_no DESC
C)
FROM dept_emp
WHERE emp_no BETWEEN 100001 AND 100003;
+-----+
| GROUP_CONCAT(DISTINCT dept_no ORDER BY emp_no DESC) |
+-----+
| d008,d005                                           |
+-----+
```

- `GROUP_CONCAT()` 을 쓸 때 주의할 점
 - 지정한 칼럼의 값들을 연결하기 위해 메모리 버퍼를 사용하는데, 지정된 버퍼의 크기를 초과하지 않게 주의해야 한다.(JDBC로 실행될 때 이 메모리 값을 초과하면 에러가 발생한다.)
 - 메모리 버퍼의 크기는 `group_concat_max_len` 시스템 변수로 조정할 수 있다. (기본값 1KB)

(9) 값의 비교와 대체

- `CASE` 문은 함수는 아니지만 유용하게 사용된다.
- `CASE WHEN ... THEN ... END` 형식으로 쿼리를 작성한다.

```
--// 서브쿼리 방식
EXPLAIN
SELECT de.dept_no, e.first_name, e.gender,
       (SELECT s.salary FROM salaries s
```

```

        WHERE s.emp_no = e.emp_no
        ORDER BY from_date DESC LIMIT 1
    ) AS last_salary
FROM dept_emp de, employees e
WHERE e.emp_no = de.emp_no
      AND de.dept_no = 'd001';

--// CASE 방식
EXPLAIN
SELECT de.dept_no, e.first_name, e.gender,
       CASE WHEN e.gender = 'F'
            THEN (SELECT s.salary FROM salaries s
                  WHERE s.emp_no = e.emp_no
                  ORDER BY from_date DESC LIMIT 1
                 )
            ELSE 0
       END AS last_salary
FROM dept_emp de, employees e
WHERE e.emp_no = de.emp_no
      AND de.dept_no = 'd001';

```

→ 위 쿼리는 서브쿼리로 작성하면 남자('M')인 경우에도 서브쿼리를 실행하여 불필요한 조회가 발생하는데, **CASE** 문을 사용하면 서브쿼리의 실행 횟수를 줄일 수 있다.

(10) 타입의 변환

- **CAST()**
 - 변환할 수 있는 데이터 타입
은 **DATE**, **TIME**, **DATETIME**, **BINARY**, **CHAR**, **DECIMAL**, **SIGNED INTEGER**, **UNSIGNED INTEGER** 이다.

```
SELECT CAST('2000-01-01' AS DATE);
```

- **CONVERT()**
 - 타입을 변환하는 용도와 문자열의 문자 집합을 변환하는 용도로 사용할 수 있다.

```
SELECT CONVERT('ABC' USING 'utf8mb4');
```

(11) 이진값과 16진수 문자열 변환

- `HEX()` : 이진값을 사람이 읽을 수 있는 형태의 16진수 문자열로 변환
- `UNHEX()` : 16진수 문자열을 읽어서 이진값으로 변환

(12) 암호화 및 해시 함수

- MD5와 SHA 모두 비대칭형 암호화 알고리즘인데, 인자로 전달한 문자열을 각각 지정된 비트 수의 해시 값을 만들어 낸다.
 - `MD5()`
 - `SHA()`
 - `SHA2()`
- 이 함수들의 결과값은 중복 가능성이 매우 낮기 때문에 길이가 긴 데이터의 크기를 줄여서 인덱싱(해시)하는 용도로도 사용된다.
- 또한, MySQL 8.0 버전부터는 함수 기반의 인덱스를 생성하면 별도 칼럼을 추가하지 않아도 된다.

```
-- 함수 기반의 인덱스 테이블 생성
CREATE TABLE tb_accesslog(
    access_id      BIGINT          NOT NULL AUTO_INCREMENT,
    access_url     VARCHAR(1000)   NOT NULL,
    access_dttm    DATETIME        NOT NULL,
    PRIMARY KEY(access_id),
    INDEX ix_accessurl( (MD5(access_url)) )
);
```

→ 이 저장공간을 더 줄이려면 MD5() 함수의 결과를 UNHEX() 함수를 사용해 이진값으로 만들면 된다.

```
{생략} INDEX ix_accessurl( ( (UNHEX(MD5(access_url))) )
```

(13) 처리 대기

- SQL 개발이나 디버깅 용도로 잠깐 대기하거나 일부러 쿼리의 실행 시간을 오랜 시간 유지하고자 할 때 상당히 유용한 함수이다.

```
SELECT SLEEP(2) FROM employees WHERE emp_no = 10001;
```

(14) 벤치마크

- `SLEEP()` 함수와 같이 디버깅이나 간단한 함수의 성능 테스트용으로 아주 유용한 함수이다.
→ 하지만, `BENCHMARK()` 함수를 사용하는 것과 실제 클라이언트 도구를 사용해서 지정된 횟수만큼 쿼리를 실행하는 것의 성능은 같지 않다. 클라이언트 도구로 쿼리를 실행할 때는 매번 쿼리의 파싱이나 최적화, 테이블 잠금이나 네트워크 비용이 소요되기 때문이다.
즉, `BENCHMARK()` 함수로 얻은 쿼리나 함수의 성능은 그 자체로는 큰 의미가 없으며, 두 개의 동일 기능을 상대적으로 비교 분석하는 용도로 사용할 것을 권장한다.

(15) IP 주소 변환

- IPv4 주소를 문자열이 아닌 부호 없는 정수 타입에 저장한다.
 - `INET_ATON()`
 - `INET_NTOA()`
- IPv6 주소를 문자열이 아닌 부호 없는 정수 타입에 저장한다.
 - `INET6_ATON()`
 - `INET6_NTOA()`

```
-- IPv6 주소를 16진수 문자 값으로 변환
SELECT HEX(INET6_ATON('fdfe::5a55:caff:fefa:9089'));
+-----+
| HEX(INET6_ATON('fdfe::5a55:caff:fefa:9089')) |
+-----+
| FDFE000000000000005A55CAFFFEFA9089          |
+-----+

-- 6진수 문자 값을 IPv6 주소로 변환
SELECT INET6_NTOA(UNHEX('FDFE000000000000005A55CAFFFEFA9089'));
+-----+
+
| INET6_NTOA(UNHEX('FDFE000000000000005A55CAFFFEFA9089')) |
+-----+
+
```

```
| fdfe::5a55:caff:fefa:9089
|
+-----+
+
```

(16) JSON 포맷

- `JSON_PRETTY()`: JSON 칼럼의 값을 읽기 쉬운 포맷으로 변환

(17) JSON 필드 크기

- `JSON_STORAGE_SIZE()`
 - JSON 데이터는 텍스트 기반이지만 MySQL 서버는 디스크의 저장 공간을 절약하기 위해 **실제 디스크에 저장할 때 BSON(Binary JSON) 포맷을 사용한다.**

(18) JSON 추출

- `JSON_EXTRACT()`
- `JSON_UNQUOTE()`

```
-- JSON 칼럼의 값을 추출
SELECT emp_no, JSON_EXTRACT(doc, "$.first_name")
FROM employee_docs;
+-----+-----+
| emp_no | JSON_EXTRACT(doc, "$.first_name") |
+-----+-----+
| 10001  | "Georgi"                          |
| 10002  | "Bezalel"                         |
| 10003  | "Parto"                           |
| 10004  | "Chirstian"                       |
| 10005  | "Kyoichi"                         |
+-----+-----+

-- JSON 칼럼의 값을 추출한 결과 값에서 " 제외
SELECT emp_no, JSON_UNQUOTE(JSON_EXTRACT(doc, "$.first_n
ame"))
FROM employee_docs;
+-----+-----+
---+
```

```

| emp_no | JSON_UNQUOTE(JSON_EXTRACT(doc, "$.first_name")) |
+-----+-----+
---+
| 10001 | Georgi |
|
| 10002 | Bezalel |
|
| 10003 | Parto |
|
| 10004 | Chirstian |
|
| 10005 | Kyoichi |
|
+-----+-----+
---+

```

- 편의성을 위해 JSON 연산자를 제공한다.

- `>` : `JSON_EXTRACT()`
- `>>` : `JSON_UNQUOTE(JSON_EXTRACT())`

```

-- JSON 칼럼의 값을 추출하기 위한 연산자 사용
SELECT emp_no, doc -> "$.first_name"
FROM employee_docs LIMIT 2;

```

(19) JSON 오브젝트 포함 여부 확인

- `JSON_CONTAINS()` : 지정된 JSON 경로에 JSON 필드를 가지고 있는지 확인

(20) JSON 오브젝트 생성

- `JSON_OBJECT()` : RDBMS 칼럼의 값을 이용해 JSON 오브젝트를 생성

```

--// JSON 오브젝트 생성
SELECT JSON_OBJECT(
    "empNo", emp_no,
    "salary", salary,
    "fromDate", from_date,

```

```

        "toDate", to_date
    ) AS as_json
FROM salaries LIMIT 1;
+-----+
-----+
| as_json
|
+-----+
-----+
| {"empNo": 10001, "salary": 60117, "toDate": "1987-06-26", "fromDate": "1986-06-26"} |
+-----+
-----+

```

(21) JSON 칼럼으로 집계

- **GROUP BY** 절과 함께 사용되는 집계 함수로서, RDBMS 칼럼의 값들을 모아 JSON 배열 또는 문서를 생성한다.
 - **JSON_OBJECTAGG()**
 - **JSON_ARRAYAGG()**

```

-- JSON 오브젝트 형식으로 집계
SELECT dept_no, JSON_OBJECTAGG(emp_no, from_date) AS agg_manager
FROM dept_manager
WHERE dept_no = 'd001'
GROUP BY dept_no;
+-----+-----+
-----+
| dept_no | agg_manager
|
+-----+-----+
-----+
| d001    | {"110022": "1985-01-01", "110039": "1991-10-01"} |
+-----+-----+
-----+

```

```
-- 배열 형식으로 집계
SELECT dept_no, JSON_ARRAYAGG(emp_no) AS agg_manager
FROM dept_manager
WHERE dept_no = 'd001'
GROUP BY dept_no;
```

dept_no	agg_manager
d001	[110022, 110039]

(22) JSON 데이터를 테이블로 변환

- `JSON_TABLE()`: JSON 데이터의 값들을 모아서 RDBMS 테이블을 만들어 반환

```
-- JSON 오브젝트를 테이블로 반환
SELECT e2.emp_no, e2.first_name, e2.gender
FROM employee_docs e1,
     JSON_TABLE(doc, "$" COLUMNS (emp_no INT PATH "$.emp_
no",
                                gender CHAR(1) PATH
                                "$.gender",
                                first_name VARCHAR(20)
                                PATH "$.first_name")
                ) AS e2
WHERE e1.emp_no IN (10001, 10002);
```

emp_no	first_name	gender
10001	Georgi	M
10002	Bezalel	F

