

7.3 MySQL 연산자와 내장 함수

| | |
|-------|----------------------|
| 🕒 생성일 | @2021년 7월 3일 오후 5:41 |
| ☰ 태그 | |

MySQL에서만 사용되는 연산자나 표기법이 있음

1. 리터럴 표기법

문자열 ⇒ " 홑따옴표로 표기

sql표준

'웅' '앵웅'

'웅' '앵웅'

MySQL만 가능

"웅" '앵웅'

"웅" "" '앵웅'

숫자

숫자값을 상수로 SQL에 사용할 때는 다른 DBMS와 마찬가지로 따옴표 없이 입력하면됨.



문자열값을 숫자값으로 자동으로 타입변환을 해주므로 성능의 차이를 고려해야 한다

- number_column = '1234'

예는 문자열 → 숫자로 변환해서 비교. 상수값 하나만 변환하므로 성능과 관련된 문제가 발생하지 않음

- string_column = 1234

문자열컬럼을 숫자로 변환해서 비교. 그러면 원래 있는 인덱스를 이용하지 못한다

그냥, 숫자값은 숫자 타입의 컬럼에만 저장해야함.

날짜



MySQL에서는 정해진 형태의 날짜 포맷으로 표기하면 MySQL서버가 자동으로 DATE나 DATETIME 값으로 변환해줌!!

```
SELECT * FROM dept_emp WHERE from_date='2011-04-29';  
SELECT * FROM dept_emp WHERE from_date=STR_TO_DATE('2011-04-29','%Y-%m-%d');
```

둘의 성능은 차이가 없다

불리언

BOOL이나 BOOLEAN이라는 타입 == TINYINT 타입

```
CREATE TABLE tb_boolean (bool_value BOOLEAN);  
INSERT INTO tb_boolean VALUES (FALSE);  
SELECT * FROM tb_boolean WHERE bool_value=FALSE;  
SELECT * FROM tb_boolean WHERE bool_value=TRUE;
```

위의 쿼리에서 TRUE나 FALSE로 비교했지만 실제 값을 조회해보면 0또는 1값이 조회됨.

그래서 불리언 타입을 꼭 사용하고 싶다면 ENUM타입으로 관리하는것이 좀 더 명확하고 실수할 가능성도 적음..

2. MySQL 연산자

동등(Equal) 비교 (=, <=>)

동등 비교는 다른 DBMS에서와 마찬가지로 "=" 기호를 사용하면 된다.

여기에 NULL값에 대한 비교까지 수행하는 기호 **<=>** : NULL - Safe 비교 연산자

→ NULL을 하나의 값으로 인식하고 비교하는 방법이다.

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

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

요렇게 널값끼리도 비교가 가능하다

부정(Not-Equal) 비교 (<>, !=)

.. <> 을 쓰는것이 가독성이 좋다.

NOT 연산자(!)

둘 다 숫자나 문자열 표현식에서도 사용할 수 있지만 부정의 결과값을 정확히 예측할 수 없는 경우에는 사용을 자제하는 것이 좋다.

```
mysql> SELECT ! 1;
```

```
+-----+
| 0 |
+-----+
```

```
mysql> SELECT !FALSE;
```

```
+-----+
| 1 |
+-----+
```

AND(&&)와 OR(||) 연산자

MySQL에서는 && 과 || 의 사용도 허용하고 있다

오라클에서는 || 을 concatenation으로 쓰고 있는데 MySQL에서도 그렇게 쓰고 싶다면 PIPE_AS_CONCAT을 설정하면됨...

(그래도 가독성을 위해 지양하자)

나누기(/, DIV)와 나머지(% , MOD) 연산자

이런것도 있따..

REGEXP 연산자

RLIKE는 정규 표현식을 비교하는 연산자..

```
mysql> SELECT 'Ø' REGEXP ' ' AS result;
+-----+
|      0 |
+-----+

mysql> SELECT 'Ø' REGEXP '[ ]' AS result;
+-----+
|      1 |
+-----+
```

REGEXP 연산자를 문자열 칼럼 비교에 사용할때 REGEXP 조건의 비교는 인덱스 레인지 스캔을 사용할 수 없다. ⇒ WHERE 조건절로 REGEXP 단독사용은 성능상 안 좋음

LIKE 연산자

인덱스 레인지 스캔이 가능한 LIKE 연산자를 더 많이 사용하자

LIKE 연산자에서 와일드카드 문자인 (%,_)가 검색어의 뒤쪽에 있다면 인덱스 레인지 스캔으로 사용할 수 있지만 앞에 있으면 사용 못함

- first_name LIKE 'Sim%' : 가능
- last_name LIKE '%Sim%' : 불가능

BETWEEN 연산자

≤ 와 ≥ 두 개의 연산자를 하나로 합친 연산자이다.

근데 index (dept_no+emp_no)칼럼으로 인덱스가 생성돼 있다.

```
SELECT * FROM dept_emp
WHERE dept_no='d003' AND emp_no=10001;
```

```
SELECT * FROM dept_emp
WHERE dept_no BETWEEN 'd003' AND 'd005' AND emp_no=10001;
```

첫번째 쿼리 : dept_no와 emp_no 조건 모두 인덱스를 이용해 범위를 줄여주는 방법으로 사용 가능

두번째 쿼리 : 모든 인덱스의 범위를 검색해야만한다. ⇒ 결국 두번째 조건문은 비교 범위를 줄이는 역할을 못함

IN 연산자와의 차이점

IN 연산자의 처리 방법은 동등 비교(=) 연산 여러개를 하나로 묶은것과 같음

→ IN과 동등 비교 연산자는 같은 형태로 인덱스를 사용하게 된다

| dept_no | emp_no |
|---------|--------|
| d002 | 499998 |
| d003 | 10005 |
| d003 | 10013 |
| d003 | 499992 |
| d004 | 10003 |
| d004 | 10004 |
| d004 | 499999 |
| d005 | 10001 |
| d005 | 10006 |
| d005 | 10006 |
| d005 | 499997 |
| d006 | 10009 |

[그림 7-2] BETWEEN(왼쪽)과 IN(오른쪽)의 인덱스 사용 방법의 차이

- BETWEEN 조건을 사용하는 위의 쿼리는 dept_emp 테이블의 (dept_no+emp_no) 인덱스의 상당히 많은 레코드를 읽게된다. 하지만 실제 가져오는 데이터는 1건....
- IN 연산자를 쓰면 emp_no = 10001 조건도 작업 범위를 줄이는 용도로 인덱스를 이용할 수 있게 된다 (동등 비교를 여러번 수행하는 것과 같은 효과를 내기 때문)

```
SELECT * FROM dept_emp USE INDEX(PRIMARY)
      WHERE dept_no BETWEEN 'd003' AND 'd005' AND emp_no=10001;

SELECT * FROM dept_emp USE INDEX(PRIMARY)
      WHERE dept_no IN ('d003', 'd004', 'd005') AND emp_no=10001;
```

인덱스 앞쪽에 있는 칼럼의 선택도가 떨어질때, IN으로 쿼리의 성능을 개선할 수도 있다.

다음은 BETWEEN 연산자를 사용하는 첫 번째 예제 쿼리의 실행 계획이다.

| id | select_type | table | type | key | key_len | ref | Rows | Extra |
|----|-------------|----------|-------|---------|---------|-----|-------|-------------|
| 1 | SIMPLE | dept_emp | range | PRIMARY | 16 | | 77140 | Using where |

그리고 다음은 BETWEEN 대신 IN 연산자를 사용한 두 번째 예제 쿼리의 실행 계획이다.

| id | select_type | table | type | key | key_len | ref | rows | Extra |
|----|-------------|----------|-------|--------|---------|-----|------|-------------|
| 1 | SIMPLE | dept_emp | range | emp_no | 4 | | 3 | Using where |

둘다 index range scan을 하고 있지만

read 하는 rows 수의 차이가 7만개와 3개 레코드로 큰 차이가 있다.



BETWEEN 비교를 사용한 쿼리에서는 부서 번호가 d003인 레코드부터 d005까지의 값을 갖고있는 모든 레코드를 다 비교하지만 IN을 사용하면 세 조합의 레코드만 비교하면 되기 때문...

BUT IN (subquery) 로 변경하는 것은 더 나쁜 결과를 가져올 수도 있다.

BETWEEN과 $\leq \geq$ 의 성능차이?!

| | emp_no BETWEEN 10001 AND 400000 | emp_no>=10001 AND emp_no<=400000 |
|----------|---------------------------------|----------------------------------|
| 평균 소요 시간 | 0.38 초 | 0.41 초 |

CPU의 연산차이에서 BETWEEN이 조금 더 빠르다

하지만 이 차이가 디스크로부터 읽어야 할 레코드 수가 달라질 정도의 차이를 만들어 내지는 않는다

IN 연산자가 상당히 비효율적으로 처리될 때

1. IN의 입력으로 서브쿼리가 사용될 때



IN의 입력으로 서브쿼리를 사용할 때는 서브쿼리가 먼저 실행되어 그 결과값이 IN의 상수 값으로 전달되는 것이 아니라 (!!!!!!!)

서브 쿼리의 **외부가 먼저 실행** 되고 IN(subquery)는 **체크 조건** 으로 사용된다.

2. NOT IN 연산자

부정형 비교라서 인덱스를 이용해 처리 범위를 줄이는 조건으로는 사용할 수 없기 때문

3. MySQL 내장 함수

NULL값 비교 및 대체 (IFNULL, ISNULL)


```
mysql> SELECT IFNULL(NULL, 1);
+-----+
|          1 |
+-----+

mysql> SELECT IFNULL(0, 1);
+-----+
|          0 |
+-----+

mysql> SELECT ISNULL(0);
+-----+
|          0 |
+-----+

mysql> SELECT ISNULL(1/0);
+-----+
|          1 |
+-----+
```

현재 시각 조회(NOW, SYSDATE)

```
mysql> SELECT NOW(), SLEEP(2), NOW();
+-----+-----+-----+
| NOW()          | SLEEP(2) | NOW()          |
+-----+-----+-----+
| 2011-04-30 17:24:03 |          0 | 2011-04-30 17:24:03 |
+-----+-----+-----+

mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();
+-----+-----+-----+
| SYSDATE()      | SLEEP(2) | SYSDATE()      |
+-----+-----+-----+
| 2011-04-30 17:24:10 |          0 | 2011-04-30 17:24:12 |
+-----+-----+-----+
```

SYSDATE() 함수 실제 함수가 수행된 딜레이 시간까지 다 적용이되어 리턴을 한다

- MySQL의 슬레이브 DB에서 안전하게 복제가 못된다

- SYSDATE()와 비교되는 칼럼은 인덱스를 효율적으로 사용하지 못한다

함수가 호출될 때마다 다른 값을 반환하므로 사실 상수가 아니다.

→ 인덱스를 스캔할때도 매번 비교되는 레코드마다 함수를 실행함



NOW()함수는 쿼리가 실행되는 시점에서 실행되고 값을 할당받아서 그 값을 SQL 문장의 모든 부분에서 사용하게 되기 때문에 쿼리가 1시간 동안 실행되더라도 항상 같은 값을 보장

날짜와 시간의 포맷 (DATE_FORMAT, STR_TO_DATE)

표준형태(년-월-일 시:분:초)로 입력된 문자열은 필요한 경우 자동적으로 DATETIME 타입으로 변환되어 처리됨

이 외에는 명시적으로 STR_TO_DATE()함수를 이용해 문자열을 DATETIME으로 변환하자

VALUES()

이 함수는 INSERT INTO ... ON DUPLICATE KEY UPDATE ...형태의 SQL 문장에서만 사용할 수 있다.



프라이머리 키나 유니크 키가 중복되는 경우에는 UPDATE를 수행하고 그렇지 않으면 INSERT를 실행

```
INSERT INTO tab_statistics (member_id, visit_count)
SELECT member_id, COUNT(*) AS cnt
FROM tab_accesslog GROUP BY member_id
ON DUPLICATE KEY
UPDATE visit_count = visit_count + VALUES(visit_count);
```

VALUES()함수를 사용하면 해당 칼럼에 INSERT 했던 값을 UPDATE 절에서 참조하는 것이 가능

COUNT(*)

- 직접 데이터나 인덱스를 읽어야만 레코드 건수를 가져올 수 있기 때문에 큰테이블에서 사용을 주의하자

| COUNT(*)쿼리에서 가장 많이 하는 실수

⇒ ORDER BY 구문이나 LEFT JOIN과 같은 레코드 건수를 가져오는 것과는 전혀 무관한 작업을 포함하는 것

⇒ SELECT 쿼리를 그대로 복사해서 칼럼이 명시된 부분만 삭제 하는것은 페이징해서 데이터를 가져오는 쿼리보다 몇배, 몇십배 더 느리게 실행될 수 있다.