DML

DATA UPDATE

INSERT

- Syntax 1

```
INSERT INTO  테이블명 [(컬럼명_리스트)]
VALUES       (컬럼값_리스트);
```

  - 컬럼명과 컬럼값은 순서대로 1:1 매핑
  - 컬럼값의 표현
    - 컬럼의 데이터유형이 문자 : ''으로 값을 표현
    - 컬럼의 데이터유형이 숫자 : ''를 **사용안함**

- Syntax 2

```
INSERT INTO  테이블명 [(컬럼명_리스트)]
SELECT 문;
```

```
INSERT INTO COMPUTER (SNO, SNAME, YEAR)
SELECT      SNO, SNAME, YEAR
FROM        STUDENT
WHERE       DEPT = '컴퓨터';
```

DELETE

- Syntax

```
DELETE FROM  테이블명
[WHERE       조건식];
```

  - WHERE절이 없으면 전체 테이블 데이터 삭제
- 전체 테이블을 삭제하는 경우, 시스템 부하가 적은 **TRUNCATE TABLE** 권장
- DDL과 DML의 차이
  - DDL : 하드디스크에서 테이블에 직접 적용, 즉시 완료
  - DML : 테이블을 메모리 버퍼에 올려놓고 작업, 실시간 영향 X, COMMIT 명령어로 트랜잭션 종료해야 반영
  - SQL Server에서는 DML도 즉시완료, COMMIT 필요없음

UPDATE

- Syntax

```
UPDATE 테이블명
SET    {컬럼명 = 산술식,}+
[WHERE 조건식];
```

```
UPDATE STUDENT
SET    YEAR = 2
WHERE  SNO = 300;
```

QUERY

SELECT

- Syntax

```
SELECT     [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }   PROJECT 연산자
[FROM      테이블_리스트]                                      JOIN 연산자
[WHERE     투플조건식]                                         SELECT 연산자
[GROUP BY  컬럼명 [HAVING 그룹조건식]]
[ORDER BY  {컬럼명|컬럼_별명|컬럼_순서 [ASC|DESC],}+];

☞ 조건식 := 컬럼명  비교연산자|SQL연산자  {숫자|문자|표현식}|컬럼명
```

  - From 절은 일부 벤더(MySQL, SQL Server)의 경우 생략 가능
- 실행순서

```
5. SELECT [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
1. FROM 테이블_리스트
2. [WHERE 투플조건식]
3. [GROUP BY  컬럼명]
4. [HAVING 그룹조건식]]
6. [ORDER BY  {컬럼명|컬럼_별명|컬럼_순서 [ASC|DESC],}+];
```

  - FROM : 실행 대상 테이블 참조
  - WHERE : 조건에 맞는 튜플만 선택
  - GROUP BY : 정해진 컬럼의 값에 따라 그룹화
  - HAVING : 각 그룹별로 그룹 조건에 맞는 튜플만 다시 선택
  - SELECT : 선택된 튜플에서 기술된 컬럼 / 표현식만 출력 / 계산 한다
  - ORDER BY : 튜플을 정렬
  - Query Optimizer가 SQL 문장의 에러를 점검하는 순서와 동일

SELECT 절

```
SELECT     [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }   PROJECT 연산자
[FROM      테이블_리스트];
```

- SELECT ALL이 디폴트 값

- DISTINCT : 중복값은 하나만 출력
- 와일드카드 문자(* 등) 사용가능
- AS(Alias)는 컬럼에 별명을 부여할 때 사용, 조건에는 사용할 수 없음 (오직 출력용)

```
SELECT   PLAYER_NAME AS 선수명, POSITION AS 위치, HEIGHT AS 키,
         WEIGHT AS 몸무게
FROM     PLAYER;
```

```
SELECT   PLAYER_NAME 선수명, POSITION 위치, HEIGHT 키,
         WEIGHT 몸무게
FROM     PLAYER;
```

  - 컬럼 별명에 공백을 넣을려면 작은 따옴표로 감싸줘야 함
- 산술연산자 사용가능

```
SELECT   PLAYER_NAME 이름,
         ROUND( WEIGHT / ((HEIGHT/100)*(HEIGHT/100)), 2) 'BMI 비만지수'
FROM     PLAYER;
```

- 합성연산자 : 컬럼과 컬럼 / 문자열을 연결해 새로운 컬럼 생성
  - Oracle : || 또는 CONCAT(String1, String2)
  - SQL Server : + 또는 CONCAT(String1, String2)
  - MySQL : ' '(Space) 또는 CONCAT(String1, String2, ..., StringN)
    - 스트링은 공백으로 연결할 수 있지만 컬럼명은 안됨

WHERE 절

- FTS(Full Table Scan) : WHERE 절이 없는 SELECT 문
  - 시스템에 매우 큰 부담이 되고, SQL 튜닝의 1차적 검토 대상이 됨
- 역할
  - 전체 튜플들에서 조건에 맞는 튜플들만 선택
  - 집단함수는 올 수 없음
- 일반형식

PROJECT 연산자

```
SELECT      [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
FROM        테이블_리스트
[WHERE      컬럼명  비교연산자|SQL연산자  {숫자|문자|표현식}|컬럼명 ];
```

SELECT 연산자

- 종류

구분	연산자	연산자의 의미
비교 연산자	=	같다.
	〉	보다 크다.
	〉=	보다 크거나 같다.
	〈	보다 작다.
	〈=	보다 작거나 같다.
SQL 연산자	BETWEEN a AND b	a와 b의 값 사이에 있으면 된다.(a와 b 값이 포함됨)
	IN (list)	리스트에 있는 값 중에서 어느 하나라도 일치하면 된다.
	LIKE '비교문자열'	비교문자열과 형태가 일치하면 된다.(%, _ 사용)
	IS NULL	NULL 값인 경우
논리 연산자	AND	앞에 있는 조건과 뒤에 오는 조건이 참(TRUE)이 되면 결과도 참(TRUE)이 된다. 즉, 앞의 조건과 뒤의 조건을 동시에 만족해야 한다.
	OR	앞의 조건이 참(TRUE)이거나 뒤의 조건이 참(TRUE)이 되어야 결과도 참(TRUE)이 된다. 즉, 앞뒤의 조건 중 하나만 참(TRUE)이면 된다.
	NOT	뒤에 오는 조건에 반대되는 결과를 되돌려 준다.
부정 비교 연산자	!=	같지 않다.
	^=	같지 않다.
	◇	같지 않다.(ISO 표준, 모든 운영체제에서 사용 가능)
	NOT 칼럼명 =	~와 같지 않다.
	NOT 칼럼명 〉	~보다 크지 않다.
부정 SQL 연산자	NOT BETWEEN a AND b	a와 b의 값 사이에 있지 않다. (a, b 값을 포함하지 않는다)
	NOT IN (list)	list 값과 일치하지 않는다.
	IS NOT NULL	NULL 값을 갖지 않는다.

- 비교연산자, SQL 연산자가 논리 연산자보다 우선 처리
- [NOT] IN (list)

```
SELECT  ENAME, JOB, DEPTNO
FROM    EMP
WHERE   (JOB, DEPTNO) IN (('MANAGER',20), ('CLERK',30));
```

```
SELECT   ENAME, JOB, DEPTNO
FROM     EMP
WHERE    JOB IN ('MANAGER','CLERK') AND DEPTNO IN (20,30);

☞        (JOB = 'MANAGER' OR JOB = 'CLERK') AND
         (DEPTNO = 20 OR DEPTNO = 30);
```

ENAME	JOB	DEPTNO
SMITH	CLERK	20
JONES	MANAGER	20
BLAKE	MANAGER	30
ADAMS	CLERK	20
JAMES	CLERK	30

5개의 행이 선택되었다.

- [NOT] LIKE
  - =의 의미
  - 와일드카드 사용가능
    - % : 0개 문자 이상의 임의의 문자열
    - _ : 1개의 단일 문자

```
SELECT   PLAYER_NAME 선수이름, POSITION 포지션, BACK_NO 백넘버,
         HEIGHT 키
FROM     PLAYER
WHERE    PLAYER_NAME LIKE '장%';
```

  - 장으로 시작하는 NAME을 가진 튜플들
- BETWEEN a AND b : 범위 설정
- IS [NOT] NULL
  - NULL(ASCII 00)
    - 값이 존재하지 않음을 표현
    - 어떤 값과도 비교 불가능(크다 / 작다 판별 불가능, 비교자체가 불가능)
  - **NULL의 특징**
    - NULL값과의 수치연산은 NULL값 리턴
    - NULL값과의 비교연산은 FALSE값 리턴
    - NULL값과의 비교연산은 IS NULL, IS NOT NULL이라는 문구를 사용해야 제대로 된 결과를 얻음

```
SELECT   PLAYER_NAME 선수이름, POSITION 포지션, TEAM_ID
FROM     PLAYER
WHERE    POSITION IS NULL;
```

- 논리 연산자
  - (), NOT, AND, OR 순으로 처리

Pseudo Column

- Oracle - ROWNUM
  - SQL 처리 결과 각 행에 대해 임시로 부여되는 일련번호
  - 결과 집합으로 출력되는 **행의 개수를 제한**할 때 주로 사용

```
SELECT   PLAYER_NAME
FROM     PLAYER
WHERE    ROWNUM <= 5;
```

- SQL Server - TOP
  - 결과 집합으로 출력되는 **행의 개수를 제한**

```
TOP(expr)  [PERCENT]  [WITH TIES]
```

```
SELECT   TOP(1) PLAYER_NAME
FROM     PLAYER
```

```
SELECT   TOP(N) PLAYER_NAME
FROM     PLAYER
```

- MySQL - LIMIT
  - 결과 집합으로 출력되는 **행의 개수를 제한**

```
SELECT   PLAYER_NAME
FROM     PLAYER
LIMIT    5;
```

SQL 단일형 내장 함수

- Built-in function(내장 함수)
  - 데이터 값을 간편하게 조작하고, SQL을 더욱 강력하게
  - 벤더별로 가장 큰 차이를 보임, 핵심적인 기능은 이름/표기법이 달라도 공통적으로 제공
- 종류
  - 단일행 함수 : 함수의 입력이 단일 행 / 문자형, 숫자형, 날짜형, 변화형, NULL 관련 함수
    - SELECT / WHERE / ORDER BY 절에 사용 가능
    - 각 행들에 대해 개별적으로 작용해 데이터 값 조작, 각각의 행에 대한 조작결과 리턴
    - 여러 인자를 입력해도 **단 하나의 결과**만 리턴
    - 함수의 인자로 상수, 변수, 표현식이 사용가능, 여러개의 인수를 가질 수 있음
    - 특별한 경우가 아니면 함수의 인자로 함수를 사용할 수 있음

종류	내용	함수의 예
문자형 함수	문자를 입력하면 문자나 숫자 값을 반환한다.	LOWER, UPPER, SUBSTR/SUBSTRING, LENGTH/LEN, LTRIM, RTRIM, TRIM, ASCII,
숫자형 함수	숫자를 입력하면 숫자 값을 반환한다.	ABS, MOD, ROUND, TRUNC, SIGN, CHR/CHAR, CEIL/CEILING, FLOOR, EXP, LOG, LN, POWER, SIN, COS, TAN
날짜형 함수	DATE 타입의 값을 연산한다.	SYSDATE/GETDATE, EXTRACT/DATEPART, TO_NUMBER(TO_CHAR(d,'YYYY'|'MM'|'DD')) / YEAR|MONTH|DAY
변환형 함수	문자, 숫자, 날짜형 값의 데이터 타입을 변환한다.	TO_NUMBER, TO_CHAR, TO_DATE / CAST, CONVERT
NULL 관련 함수	NULL을 처리하기 위한 함수	NVL/ISNULL, NULLIF, COALESCE

※ 주: Oracle함수/SQL Server함수 표시, '/' 없는 것은 공통 함수

- 다중행 함수 : 입력이 여러 행, 하나의 결과만 리턴
  - 집단 함수 : COUNT, SUM, AVG, MIN, MAX, STDDEV
  - 그룹 함수 : ROLLUP, CUBE, GROUPING SETS
  - 윈도우 함수 : 다양한 분석 가능

- Syntax

```
함수명 ( 컬럼 | 표현식 [, arg1, arg2, ...])
```

문자형 함수

문자형 함수	함수 설명
LOWER(문자열)	문자열의 알파벳 문자를 소문자로 바꾸어 준다.
UPPER(문자열)	문자열의 알파벳 문자를 대문자로 바꾸어 준다.
ASCII(문자)	문자나 숫자를 ASCII 코드 번호로 바꾸어 준다.
CHR/CHAR(ASCII번호)	ASCII 코드 번호를 문자나 숫자로 바꾸어 준다.
CONCAT (문자열1, 문자열2)	Oracle, My SQL에서 유효한 함수이며 문자열1과 문자열2를 연결한다. 합성 연산자'||'(Oracle)나 '+'(SQL Server)와 동일하다.
SUBSTR/SUBSTRING (문자열, m[, n])	문자열 중 m위치에서 n개의 문자 길이에 해당하는 문자를 돌려준다. n이 생략되면 마지막 문자까지이다.
LENGTH/LEN(문자열)	문자열의 개수를 숫자값으로 돌려준다.
LTRIM (문자열 [, 지정문자])	문자열의 첫 문자부터 확인해서 지정 문자가 나타나면 해당 문자를 제거한다. (지정 문자가 생략되면 공백 값이 디폴트) SQL Server에서는 LTRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.
RTRIM (문자열 [, 지정문자])	문자열의 마지막 문자부터 확인해서 지정 문자가 나타나는 동안 해당 문자를 제거한다.(지정 문자가 생략되면 공백 값이 디폴트) SQL Server에서는 LTRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.
TRIM ([leading|trailing|both] 지정문자 FROM 문자열)	문자열에서 머리말, 꼬리말, 또는 양쪽에 있는 지정 문자를 제거한다. (leading | trailing | both 가 생략되면 both가 디폴트) SQL Server에서는 TRIM 함수에 지정문자를 사용할 수 없다. 즉, 공백만 제거할 수 있다.

- Oracle - DUAL 테이블
  - 사용자 SYS가 소유하며, 모든 사용자가 액세스 가능한 테이블
  - SELECT ~ FROM ~의 형식을 갖추기 위한 일종의 **더미 테이블**
  - DUMMY라는 VARCHAR2(1) 유형의 칼럼에 'X'라는 값이 들어 있는 행을 1건 포함
- SQL Server, MySQL은 SELECT절 만으로도 문장이 가능하므로 더미 테이블이 필요없음

숫자형 함수

숫자형 함수	함수 설명
ABS(숫자)	숫자의 절대값을 돌려준다.
SIGN(숫자)	숫자가 양수인지, 음수인지 0인지를 구별한다.
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나누어 나머지 값을 리턴한다. MOD 함수는 % 연산자로도 대체 가능함 (ex:7%3)
CEIL/CEILING(숫자)	숫자보다 크거나 같은 최소 정수를 리턴한다.
FLOOR(숫자)	숫자보다 작거나 같은 최대 정수를 리턴한다.
ROUND(숫자 [, m])	숫자를 소수점 m자리에서 반올림하여 리턴한다. m이 생략되면 디폴트 값은 0이다.
TRUNC(숫자 [, m])	숫자를 소수 m자리에서 잘라서 버린다. m이 생략되면 디폴트 값은 0이다. SQL SERVER에서 TRUNC 함수는 제공되지 않는다.
SIN, COS, TAN,...	숫자의 삼각함수 값을 리턴한다.
EXP(), POWER(), SQRT(), LOG(), LN()	숫자의 지수, 거듭 제곱, 제곱근, 자연 로그 값을 리턴한다.

```
SELECT  ENAME, ROUND(SAL/12,1), TRUNC(SAL/12,1)
FROM    EMP;
```

- MySQL에서는 TRUNC 대신 TRUNCATE

날짜형 함수(MySQL)

- 내부적으로 숫자 형식으로 변환하여 저장
  - 공간 절약, 산술 연산이 가능
  - 출력기에는 년 / 월 / 일 / 시 / 분 / 초단위로 추출하여 변환해야 함
- DBMS마다 날짜형 함수의 기능적 차이가 **매우 심함**
- SYSDATE(), NOW()
  - 공통점
    - 컨텍스트가 스트링일 경우 : 'YYYY-MM-DD HH:MM:SS' 포맷
    - 컨텍스트가 숫자일 경우 : YYYYMMDDHHDDSS.uuuuuu 포맷
  - 차이점
    - SYSDATE() : 현재 시간
    - NOW() : 명령어가 실행된 시간
      - DATETIME, TIMESTAMP 컬럼의 **디폴트 값**을 정의하는데 사용

```
CREATE TABLE tmp (
    id              INT PRIMARY KEY AUTO_INCREMENT,
    title           VARCHAR(255) NOT NULL,
    created_on      DATETIME NOT NULL DEFAULT NOW()
                                // or CURRENT_TIMESTAMP
);
```

      - CURRENT_TIMESTAMP() / CURRENT_TIMESTAMP와 동의어
- TIMESTAMP(), DATE(), TIME()

```
SELECT   TIMESTAMP(NOW()) AS CurrentTimestamp;
SELECT   DATE(NOW()) AS CurrentDate,
SELECT   TIME(NOW()) AS CurrentTime,
```

- EXTRACT()

```
EXTRACT(unit FROM date)

☞ unit ::= QUARTER | YEAR | YEAR_MONTH | MONTH | WEEK |
           DAY | DAY_HOUR | DAY_MINUTE | DAY_SECOND | DAY_MICROSECOND |
           HOUR | HOUR_MINUTE | HOUR_SECOND | HOUR_MICROSECOND |
           MINUTE | MINUTE_SECOND | MINUTE_MICROSECOND |
           SECOND | SECOND_MICROSECOND
```

- DATEDIFF(t1, t2)
  - t1 - t2
  - 두 개의 DATE, DATETIME, TIMESTAMP **값의 차이**를 계산
  - 두 인자의 **타입이 다르면**, NULL 값을 리턴
  - DATETIME, TIMESTAMP 값의 경우 **시간 부분은 무시(날짜만** 처리)

```
SELECT   DATEDIFF('2009-03-01', '2009-01-01') diff          /* 59 */
```

- TIMEDIFF(t1, t2)
  - t1 - t2
  - 두개의 TIME, DATETIME 값의 차이를 계산
  - 범위 : -838:59:59 ~ 838:59:59 (TIME은 3바이트)

```
SELECT   TIMEDIFF('2009-02-01', '2009-01-01') diff          /* 744:00:00 */
```

```
SELECT   TIMEDIFF('2009-03-01 00:00:00', '2009-01-01 00:00:00') diff;
SHOW     WARNINGS;                                  /* 838:59:59, 범위 에러 */
```

```
SELECT   TIMESTAMPDIFF(HOUR, '2009-01-01 00:00:00',
                             '2009-03-01 00:00:00') diff;       /* 1416 */
```

- TIMESTAMPDIFF(unit, begin, end)

```
TIMESTAMPDIFF(unit, begin, end);

☞ unit ::=  MICROSECOND | SECOND | MINUTE | HOUR |
            DAY | WEEK | MONTH |
            QUARTER | YEAR
```

```
SELECT   TIMESTAMPDIFF(MINUTE, '2010-01-01 10:00:00',
                               '2010-01-01 10:45:59') result;  /* 45 */

SELECT   TIMESTAMPDIFF(SECOND, '2010-01-01 10:00:00',
                               '2010-01-01 10:45:59') result;  /* 2759 */
```

  - end - begin값을 계산

    - 앞의 두 함수와 다르게 뒤의 시간에서 앞의 시간을 뺌
- 나이계산

```
SELECT   PLAYER_NAME AS 선수명, BIRTH_DATE AS 생일,
         TIMESTAMPDIFF(YEAR, BIRTH_DATE, DATE(NOW())) AS 나이
FROM     PLAYER;
```

```
SELECT   PLAYER_NAME AS 선수명, BIRTH_DATE AS 생일,
         FLOOR(DATEDIFF(DATE(NOW()), BIRTH_DATE) / 365) AS 나이
FROM     PLAYER;
```

```
SELECT   TIMESTAMPDIFF(YEAR, '2000-08-02', '2019-05-15'),
         YEAR('2019-05-15') - YEAR('2000-08-02');                    /* 18, 19 */
```

    - TIMESTAMPDIFF의 unit으로 YEAR를 지정한 뒤 계산 하면, 정확히 달,일을 지나야 숫자가 올라감
- DATE_ADD() 및 DATE_SUB()
    - DATE / DATETIME 값에 interval을 더하거나 뺌

```
DATE_ADD(start_date, interval_expression)
DATE_SUB(start_date, interval_expression)

☞ interval_expression ::= INTERVAL expr unit
☞ unit ::= SECOND | MINUTE | HOUR |
           DAY | WEEK | MONTH |
           QUARTER | YEAR
```

```
SELECT   DATE_ADD('1999-12-31 00:00:01',
                  INTERVAL 1 DAY) result;
SELECT   DATE_ADD('1999-12-31 23:59:59',
                  INTERVAL '1:1' MINUTE_SECOND) result;
SELECT   DATE_ADD('2000-01-01 00:00:00',
                  INTERVAL '-1 5' DAY_HOUR) result;
SELECT   DATE_ADD('1999-12-31 23:59:59.000002',
                  INTERVAL '1.999999' SECOND_MICROSECOND) result;
```

    - DATE에 HOUR / MINUTE등을 더하면 DATETIME으로 자동변환됨
- TIME_ADD() 및 TIME_SUB()
    - DATE_ADD / SUB 와 개념적으로 동일

```
TIME_ADD(start_time, interval_expression)
TIME_SUB(start_time, interval_expression)

☞ interval_expression ::= INTERVAL expr unit
☞ unit ::= SECOND | MINUTE | HOUR |
           DAY | WEEK | MONTH |
           QUARTER | YEAR
```

- DATE_FORMAT()

```
DATE_FORMAT(date, format)

☞ format ::= format_specifier 들의 조합
☞ format specifier ::= %Y | %M | %D | %H | %M | %S |
                       %y | %m | %d | %h | %m | %s | . . .
```

```
SELECT   orderNumber,
         DATE_FORMAT(orderdate, '%Y-%m-%d') orderDate,
         DATE_FORMAT(requireddate, '%a %D %b %Y') requiredDate,
         DATE_FORMAT(shippedDate, '%W %D %M %Y') shippedDate
FROM     orders;
```

- GET_FORMAT()

```
GET_FORMAT( {DATE | TIME | DATETIME}, {'EUR' | 'USA' | 'JIS' | 'ISO' | 'INTERNAL'} )
```

```
SELECT   DATE_FORMAT('2003-03-31', GET_FORMAT(DATE, 'ISO')) AS ISO,
         DATE_FORMAT('2003-03-31', GET_FORMAT(DATE, 'JIS')) AS JIS,
         DATE_FORMAT('2003-03-31', GET_FORMAT(DATE, 'USA')) AS USA,
         DATE_FORMAT('2003-03-31', GET_FORMAT(DATE, 'EUR')) AS EUR,
         DATE_FORMAT('2003-03-31', GET_FORMAT(DATE, 'INTERNAL')) AS
                                                              INTERNAL;
```

```
SELECT   orderNumber,
         DATE_FORMAT(orderdate, GET_FORMAT(DATE,'ISO') ) orderDate,
         DATE_FORMAT(requireddate, GET_FORMAT(DATE,'USA') ) requiredDate,
         DATE_FORMAT(shippedDate, GET_FORMAT(DATE,'EUR') ) shippedDate
FROM     orders;
```

- STR_TO_DATE()

```
STR_TO_DATE(string, format)

☞ format ::= specifier 들의 조합
☞ specifier ::= %Y | %M | %D | %H | %M | %S |
                %y | %m | %d | %h | %m | %s | . . .
```

```
SELECT   STR_TO_DATE('21,5,2013', '%d,%m,%Y'); /* 2013-05-21 */
SELECT   STR_TO_DATE('2013', '%Y');            /* 2013-00-00 */
SELECT   STR_TO_DATE('113005', '%h%i%s');      /* 11:30:05 */
SELECT   STR_TO_DATE('11', '%h');              /* 11:00:00 */
SELECT   STR_TO_DATE('20130101 1130', '%Y%m%d %h%i') ;
                                               /* 2013-01-01 11:30:00 */
SELECT   STR_TO_DATE('21,5,2013 extra characters', '%d,%m,%Y');
                                               /* 2013-05-21 */
```

  - 스트링 타입을 데이트로 변환

변환형 함수

- CAST()
  - 데이터를 다른 데이터 타입으로 변환

```
CAST(expr AS datatype);
```

```
SELECT   CONCAT('Date: ', CAST(NOW() AS DATE));
```

```
SELECT   TEAM_ID, ZIP_CODE1, ZIP_CODE2,
         CAST(ZIP_CODE1 AS UNSIGNED) + CAST(ZIP_CODE2 AS UNSIGNED)
         AS 우편번호합
FROM     TEAM;
```

- CONVERT()
  - 데이터를 다른 캐릭터 셋으로 변환

```
CONVERT(expr USING transcoding_name);
```

```
CONVERT('test' USING utf8);
```

CASE Expression

- IF-THEN-ELSE 논리와 유사, SQL의 비교 연산기능을 보완하는 역할
- 중첩가능
- Simple Case Expression

```
CASE     expr
         {WHEN   comparison_expr   THEN     return_expr}+
         ELSE    표현식
END
```

```
SELECT   LOC,
         CASE    LOC
                 WHEN   'NEW YORK'      THEN    'EAST'
                 WHEN   'BOSTON'        THEN    'EAST'
                 WHEN   'CHICAGO'       THEN    'CENTER'
                 WHEN   'DALLAS'        THEN    'CENTER'
                 ELSE   'ETC'
         END     AS AREA
FROM DEPT;
```

- Searched Case Expression

```
CASE
         {WHEN   condition_expr     THEN     return_expr}+
         ELSE    표현식
END

☞ condition_expr ::= "expr 비교연산자 comparison_expr"
```

```
SELECT   ENAME,
         CASE     WHEN   SAL >= 3000      THEN    'HIGH'
                  WHEN   SAL >= 1000      THEN    'MID'
                  ELSE '  LOW'
         END      AS SALARY_GRADE
FROM EMP;
```

```
SELECT   PLAYER_NAME,
         CASE     POSITION
                  WHEN 'FW'       THEN 'Forward'
                  WHEN 'DF'       THEN 'Defense'
                  WHEN 'MF'       THEN 'Mid-field'
                  WHEN 'GK'       THEN 'Goal keeper'
                  ELSE '*****'
         END AS 포지션
FROM     PLAYER;
```

```
SELECT   PLAYER_NAME,
         CASE
                  WHEN POSITION = 'FW'     THEN 'Forward'
                  WHEN POSITION = 'DF'     THEN 'Defense'
                  WHEN POSITION = 'MF'     THEN 'Mid-field'
                  WHEN POSITION = 'GK'     THEN 'Goal keeper'
                  ELSE '*****'
         END AS 포지션
FROM     PLAYER;
```

- CASE에서 ELSE절을 생략하면 DEFAULT값은 NULL이 됨
- 중첩된 CASE Expression

```
SELECT   ENAME, SAL,
         CASE     WHEN   SAL >= 2000       THEN    1000
                  ELSE   (CASE    WHEN   SAL >= 1000       THEN    500
                                  ELSE   0
                          END)
         END AS BONUS
FROM     EMP;
```

- 오라클에서는 DECODE() 함수로 표현

```
DECODE (comparison_expr, return_expr1, ... return_exprn, 디폴트 수식)
```

NULL 관련 함수 : Standard 함수

- NULL 값의 특징
  - 테이블을 생성할 때 NOT NULL / PRIMARY KEY로 정의되지 않은 모든 데이터 유형은 널 값을 포함할 수 있다

    - 널 값을 포함하는 연산은 그 결과값도 널값이다
- NULL값의 처리방법
    - 테이블 출력시, NULL값은 다른 값으로 대체하는 것이 보기 편함
        - 숫자유형 : 0
        - 문자유형 : blank보다는 ****같이 해당 시스템에서 의미 없는 문자로
    - COALESCE(e1,e2,...)함수 사용
        - 오라클의 NVL(e1, e2) 함수
        - SQL Server의 ISNULL(e1,e2) 함수
        - MySQL의 IFNULL(e1,e2) 함수
            - MySQL의 ISNULL(expr)함수는 비교연산자 IS NULL을 구현한 것(NULL이면 0, 아니면 1 리턴)
        - **이식성을 위해 위의 비표준 함수들은 사용을 자제**
            - expr1이 NULL이 아니면 expr1, NULL이면 e2를 출력(일반적으로 NULL이 아닌값 지정)

COALESCE

COALESCE (expr1, expr2, ...)

- 임의 개수의 expr에서 널이 아닌 **최초의 expr**을 리턴, 모든 expr이 널이면 **널**을 리턴

```
SELECT  ENAME, COMM, SAL, COALESCE(COMM, SAL) COAL
FROM    EMP;
```

ENAME	COMM	SAL	COAL
SMITH		800	800
ALLEN	300	1600	300
WARD	500	1250	500
JONES		2975	2975
MARTIN	1400	1250	1400
BLAKE		2850	2850
CLARK		2450	2450
SCOTT		3000	3000
KING		5000	5000
TURNER	0	1500	0
ADAMS		1100	1100
JAMES		950	950
FORD		3000	3000
MILLER		1300	1300

14개의 행이 선택되었다.

- n개의 중첩된 CASE 문장으로 표현 가능함

```
SELECT  ENAME, COMM, SAL, COALESCE(COMM, SAL) COAL
FROM    EMP;
```

```
SELECT  ENAME, COMM, SAL,
        CASE    WHEN    COMM IS NOT NULL         THEN COMM
                ELSE    (CASE   WHEN    SAL IS NOT NULL  THEN SAL
                                ELSE    NULL
                        END)
        END COAL
FROM    EMP;
```

NULLIF

```
NULLIF (expr1, expr2)
```

- expr1 = expr2면 NULL, 아니면 expr1을 리턴

```
SELECT  ENAME, EMPNO, MGR, NULLIF(MGR, 7698) NUIF
FROM    EMP;
```

```
SELECT  ENAME, EMPNO, MGR,
        CASE    WHEN    MGR = 7698      THEN    NULL
                ELSE    MGR
        END NUIF
FROM    EMP;
```

GROUP BY 절

- 튜플들을 그룹별로 분류해 그룹 별 통계정보를 얻을 때 사용
- 특징
  - GROUP BY절을 통해 그룹별 기준을 정한 후, SELECT 절에 집단 함수 사용
  - 집단함수는 NULL값을 가진 행을 제외하고 수행
  - 컬럼 별명(Alias)사용 불가
  - WHERE절은 GROUP BY절보다 먼저 수행
  - 집단함수는 WHERE절에는 갈 수 없음

```
SELECT      [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
[FROM       테이블_리스트]
[WHERE      투플조건식]
[GROUP BY   컬럼명 [HAVING 그룹조건식]];
```

- 실행
  - 사용시 **그루핑 기준 컬럼** / 집단 함수에서 사용되는 **숫자형 데이터 컬럼들의 튜플집합**을 새로 만듬, 그 이외의 컬럼들은 메모리에서 제거(MySQL은 아님)

- 따라서 이들 이외의 컬럼은 SELECT 절에서 에러발생

HAVING 절

- GROUP BY절의 기준 항목이나 그룹의 집단 함수를 이용할 조건을 표시
- **GROUP BY절에 의해 만들어진 그룹들 중**, **추가의 제한조건**을 두어 이 조건을 만족하는 그룹만 출력
- GROUP BY절 뒤에 위치

```
SELECT      [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
FROM        테이블_리스트
[WHERE      투플조건식]
[GROUP BY   컬럼명 [HAVING 그룹조건식]]
[ORDER BY   {컬럼명|컬럼_별명|컬럼_순서 [ASC|DESC],}+];
```

  - WHERE 튜플 조건식 : 튜플조건식이 TRUE인 **튜플**만 선택(집단함수 사용 불가)
  - HAVING 그룹 조건식 : 그룹조건식이 TRUE인 그룹만 선택(집단함수를 이용해 서술)
- HAVING절에서 사용할 수 있는 컬럼은
  - GROUP BY절에서 사용한 컬럼
  - 집단 함수에서 사용한 컬럼
  - MySQL은 다 사용가능, SQL 표준이 위처럼 서술됨

```
SELECT   TEAM_ID 팀ID, COUNT(*) 인원수
FROM     PLAYER
WHERE    TEAM_ID IN ('K09', 'K02')
GROUP    BY TEAM_ID;

팀ID          인원수
-----         -------
K02              49
K09              49
2개의 행이 선택되었다.
```

```
SELECT   TEAM_ID 팀ID, COUNT(*) 인원수
FROM     PLAYER
GROUP    BY TEAM_ID  HAVING TEAM_ID IN ('K09', 'K02');
```

- 같은 결과를 만들지만, WHERE로 먼저 튜플수를 거르는게 효율적

예제 / 집단함수(CASE문) ~ GROUP BY ~ 형태의 활용

- 모든 부서의 월별 입사자의 평균 급여를 구하여라

```
SELECT   ENAME, DEPTNO, EXTRACT(MONTH FROM HIREDATE) 입사월, SAL
FROM     EMP;
```

```
SELECT   ENAME, DEPTNO,
         CASE MONTH WHEN 1 THEN SAL END M01,
         CASE MONTH WHEN 2 THEN SAL END M02,
         CASE MONTH WHEN 3 THEN SAL END M03,
         CASE MONTH WHEN 4 THEN SAL END M04,
         CASE MONTH WHEN 5 THEN SAL END M05,
         CASE MONTH WHEN 6 THEN SAL END M06,
         CASE MONTH WHEN 7 THEN SAL END M07,
         CASE MONTH WHEN 8 THEN SAL END M08,
         CASE MONTH WHEN 9 THEN SAL END M09,
         CASE MONTH WHEN 10 THEN SAL END M10,
         CASE MONTH WHEN 11 THEN SAL END M11,
         CASE MONTH WHEN 12 THEN SAL END M12
FROM     (
                  SELECT   ENAME, DEPTNO,
                           EXTRACT(MONTH FROM HIREDATE) MONTH, SAL
                  FROM     EMP
         );
```

```
SELECT   DEPTNO,
         AVG(CASE MONTH WHEN 1 THEN SAL END) M01,
         AVG(CASE MONTH WHEN 2 THEN SAL END) M02,
         AVG(CASE MONTH WHEN 3 THEN SAL END) M03,
         AVG(CASE MONTH WHEN 4 THEN SAL END) M04,
         AVG(CASE MONTH WHEN 5 THEN SAL END) M05,
         AVG(CASE MONTH WHEN 6 THEN SAL END) M06,
         AVG(CASE MONTH WHEN 7 THEN SAL END) M07,
         AVG(CASE MONTH WHEN 8 THEN SAL END) M08,
         AVG(CASE MONTH WHEN 9 THEN SAL END) M09,
         AVG(CASE MONTH WHEN 10 THEN SAL END) M10,
         AVG(CASE MONTH WHEN 11 THEN SAL END) M11,
         AVG(CASE MONTH WHEN 12 THEN SAL END) M12
FROM     (
                  SELECT   ENAME, DEPTNO,
                           EXTRACT(MONTH FROM HIREDATE) MONTH, SAL
                  FROM     EMP
         )
GROUP    BY DEPTNO;
```

DEPTNO	M01	M02	M03	M04	M05	M06	M07	M08	M09	M10	M11	M12
30		1425			2850				1375			950
20				2975			2050					1900
10	1300					2450					5000	

3개의 행이 선택되었다.

집단함수와 NULL 처리

- 집단함수에서는 NULL값을 갖는 튜플은 **제외하고 실행**
    - 집단함수 내에서 NVL / ISNULL 함수는 피해야함 (계산량만 증가)
    - 집단함수의 결과를 NVL / ISNULL를 사용해 디스플레이 하는건 바람직함
- CASE 절에서 ELSE를 생략하면 디폴트값이 NULL
    - 가능하면 ELSE 절의 상수값을 지정하지 않거나, ELSE절을 작성하지 않도록 함

예제 / 집단함수(CASE문) ~ GROUP BY ~ 형태의 활용

- 팀별로 포지션별 인원수, 그리고 팀별 전체 인원수를 구하라. 단 데이터가 없는 경우 0으로 표시

```
SELECT   PLAYER_NAME, TEAM_ID,
         CASE POSITION WHEN 'FW' THEN 1 ELSE 0 END FW,
         CASE POSITION WHEN 'MF' THEN 1 ELSE 0 END MF,
         CASE POSITION WHEN 'DF' THEN 1 ELSE 0 END DF,
         CASE POSITION WHEN 'GK' THEN 1 ELSE 0 END GK,
         CASE WHEN POSITION IS NULL THEN 1 ELSE 0 END UNDECIDED,
         COUNT(*) SUM
FROM     PLAYER
GROUP    BY TEAM_ID;
```

```
SELECT   TEAM_ID,
         SUM(CASE POSITION WHEN 'FW' THEN 1 ELSE 0 END) FW,
         SUM(CASE POSITION WHEN 'MF' THEN 1 ELSE 0 END) MF,
         SUM(CASE POSITION WHEN 'DF' THEN 1 ELSE 0 END) DF,
         SUM(CASE POSITION WHEN 'GK' THEN 1 ELSE 0 END) GK,
         SUM(CASE WHEN POSITION IS NULL THEN 1 ELSE 0 END) UNDECIDED,
         COUNT(*) SUM
FROM     PLAYER
GROUP    BY TEAM_ID;
```

ORDER BY 절

- 테이블을 특정 컬럼 기준으로 정렬(오름차순/내림차순)
- 컬럼명 대신 컬럼 별명이나 컬럼 순서를 나타내는 정수도 가능

```
SELECT      [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
FROM        테이블_리스트
[WHERE      투플조건식]
[GROUP BY   컬럼명 [HAVING 그룹조건식]]
[ORDER BY   {컬럼명|컬럼_별명|컬럼_순서 [ASC|DESC],}+];
```

- 특징
    - 디폴트는 오름차순(ASC)
    - GROUP BY에서는 컬럼 별명 사용불가 / ORDER BY에서는 컬럼 별명 **사용가능**
    - Oracle에서는 NULL값을 가장 큰값으로, SQL Server / MySQL에서는 가장 작은값으로 간주

```
SELECT   DNAME, LOC AREA, DEPTNO
FROM     DEPT
ORDER    BY DNAME, LOC, DEPTNO DESC;
```

```
SELECT   DNAME, LOC AREA, DEPTNO
FROM     DEPT
ORDER    BY DNAME, AREA, DEPTNO DESC;
```

```
SELECT   DNAME, LOC AREA, DEPTNO
FROM     DEPT
ORDER    BY 1, AREA, 3 DESC;
```

- 셋 모두 동일
- 주의사항
  - SELECT 목록에 나타나지 않은 문자형 컬럼이 포함될 수 있음
    - DBMS가 데이터를 메모리에 올릴 때 튜플단위로 모든 컬럼을 가져옴
    - SELECT에서 일부 컬럼만 선택해도, ORDER BY절에서 메모리에 올라와 있는 다른 컬럼의 데이터 사용 가능
  - GROUP BY 절을 같이 사용시에는 SELECT 목록에 나타나지 않은 문자형 컬럼이 ORDER BY에 포함될 수 없음
    - GROUP BY 절이 튜플의 집합을 새로 만들고 나머지는 메모리에서 제거

```
SELECT   JOB, SAL
FROM     EMP
GROUP    BY JOB    HAVING COUNT(*) > 0
ORDER    BY SAL;                          // Error !! (GROUP BY)
```

```
SELECT   JOB
FROM     EMP
GROUP    BY JOB    HAVING COUNT(*) > 0
ORDER    BY SAL;                          // Error !! (ORDER BY)
```

```
SELECT   JOB
FROM     EMP
GROUP    BY JOB    HAVING COUNT(*) > 0
ORDER    BY MAX(EMPNO), MAX(MGR), SUM(SAL), COUNT(DEPTNO),
         MAX(HIREDATE);                   // No Error !!
```

TOP-N Query

- 순위가 높은 N개의 튜플을 추출하는 질의
- 문제점
  - Oracle : ROWNUM / ORDER BY 절을 사용해 직접사용하면 잘못된 결과가 나옴
  - SELECT문은 정렬이 완료된 후 데이터의 일부가 출력되는 것이 아니라 **데이터의 일부가 먼저 추출된 후** 정렬작업이 실행됨
    - ORDER BY절이 결과 집합을 결정하는데 관여하지 않음

- 해결책
  - Oracle : ORDER BY 절을 Inline View를 이용하여 작성한 후, ROWNUM 변수 사용 / 동점자 처리는 불가
  - SQL Server : TOP(n) 함수 이용

Standard Query

- Windows functions : RANK()와 ROW_NUMBER()함수
  - RANK() : 동점자 처리
  - ROW_NUMBER() : 일련번호 부여, Pagination에 활용

```
SELECT   ROW_NUMBER( ) OVER (ORDER BY SEAT_COUNT DESC) AS ROW_NUM,
         STADIUM_ID, STADIUM_NAME, SEAT_COUNT,
         RANK( ) OVER (ORDER BY SEAT_COUNT DESC) AS SEAT_RANK
FROM     STADIUM;
```

- MySQL에서
  - RANK() 함수를 사용한 컬럼명에 RANK를 사용할 수 없음
  - ROW_NUMBER() 함수를 사용한 컬럼명에 ROW_NUMBER를 사용할 수 없음
- MySQL의 Limit Clause
  - 첫번째 파라미터 : 결과에서 리턴할 첫번째 레코드의 위치(offset)
    - 생략가능, 생략되면 디폴트 = 0
  - 두번째 파라미터 : 리턴할 레코드의 최대 수

```
SELECT [ALL|DISTINCT] {{컬럼명 [[AS] 컬럼_별명],}+ | * }
FROM 테이블_리스트
[WHERE 투플조건식]
[GROUP BY  컬럼명]
[HAVING 그룹조건식]]
[ORDER BY  {컬럼명|컬럼_별명|컬럼_순서 [ASC|DESC],}+]
[LIMIT  [offset, ] row_count];
```

```
SELECT   PLAYER_NAME, HEIGHT
FROM     PLAYER
ORDER    BY HEIGHT DESC
LIMIT    3;                              // LIMIT  0, 3
```

Oracle

- 사원에서 급여가 높은 3명만 내림차순으로 출력

```
SELECT   ENAME, SAL
FROM     EMP
WHERE    ROWNUM < 4
ORDER    BY SAL DESC;                          // Semantic error !!!
```

– 급여가 높은 3명만 출력 (inline view 이용)

```
SELECT   ENAME, SAL
FROM     (SELECT   ENAME, SAL
          FROM     EMP
          ORDER    BY SAL DESC)
WHERE    ROWNUM < 4;
```

SQL Server

- 사원 테이블에서 급여가 높은 2명만 내림차순으로 출력(TOP 함수 사용)

```
SELECT   TOP(2) ENAME, SAL
FROM     EMP
ORDER    BY SAL DESC;
```

- 동점자 처리

```
SELECT   TOP(2) WITH TIES ENAME, SAL
FROM     EMP
ORDER    BY SAL DESC;
```