

1. DQL

DQL(Data Query Language) - 테이블의 데이터를 검색하기 위해 사용하는 명령어

- DML의 하위그룹으로 분류
- 데이터 조회결과를 **Result Set**(결과집합)이라고 함
- 0행 이상의 결과집합을 리턴받음
- **Select**를 사용. **select**는 테이블에 저장된 데이터를 조회하는데 사용되는 가장 기본적인 문법이고 제일 많이 사용.

★select

- 실제 결과집합의 컬럼을 제한
- 존재하지 않는 컬럼도 조회함
- 가상컬럼(연산처리결과) 사용 가능
- 123, '안녕'같은 리터럴도 행수만큼 반복 출력함.
- 실급여 부분에서 **bonus**부분이 null이면 null로 나온다.

```
select
```

```
emp_name,
```

```
salary,
```

```
bonus,
```

```
nvl(bonus, 0),
```

```
salary * 12,
```

```
salary + (salary * nvl(bonus, 0)) as "실급여",
```

```
'안녕'
```

```
from
```

```
employee;
```

SQL | 인출된 모든 행: 24(0.02초)

EMP_NAME	SALARY	BONUS	NVL(BONUS,0)	SALARY*12	실급여	'안녕'
1 선동일	8000000	0.3	0.3	96000000	10400000	안녕
2 송종기	6000000	(null)	0	72000000	(null)	안녕
3 노웅결	3700000	(null)	0	44400000	(null)	안녕
4 송은희	2800000	(null)	0	33600000	(null)	안녕
5 유재식	3400000	0.2	0.2	40800000	4080000	안녕
6 정중하	3900000	(null)	0	46800000	(null)	안녕
7 박나라	1800000	(null)	0	21600000	(null)	안녕
8 하이유	2200000	0.1	0.1	26400000	2420000	안녕
9 김해술	2500000	(null)	0	30000000	(null)	안녕

잠깐! 여기서부터 알고가면 좋다.

코드 실제 처리 순서

`select` (필수) - 5. 조회할 컬럼

`from` (필수) - 1. 조회할 테이블

`where` - 2. 조건절(`true`-결과집합에 포함. `false`-결과집합에 제외) 기본적인 조건절로서 모든 필드를 조건에 둘 수 있음.

`group by` - 3. 행을 특정컬럼 기준으로 그룹으로 묶는것

`having` - 4. 그룹핑된 결과에 대한 조건절. 무조건 `group by`된 이후 특정한 필드로 그룹화된 새로운 테이블에 대한 조건을 줄 수 있다.

`order by` - 6. 행간 정렬(`ex`: 오름차순, 내림차순)

order by절 기본 구조

select * from 테이블명

order by 컬럼명 (asc: 오름차순, desc: 내림차순. 안쓰면 기본값은 오름차순이기 때문에 오름차순으로 정렬됨)

자세한 내용은 아래에 -> 자세한 설명이 있는 <https://gomguard.tistory.com/93>에서 들고 온 페이지 내용이다.

ORDER BY 절의 각종 사용 방법

- ORDER BY 같은 경우에는 열의 이름 뿐 아니라 ALIAS 를 입력해도 무방합니다. 이는 ORDER BY 가 쿼리문 중에 가장 나중에 실행되는 쿼리문 이기 때문입니다. 또한 열의 숫자위치로 정렬하는 것도 가능하며 여러 열을 기준으로 정렬하는 것 또한 가능합니다.
- 기본 형태
 - `SELECT * FROM table_name ORDER BY column_name;`
- ALIAS 사용
 - `SELECT sal + comm AS TOTAL FROM emp ORDER BY TOTAL;`
- 열의 숫자를 사용
 - `SELECT * FROM table_name ORDER BY 3;`
 - 3번째 열을 기준으로 정렬하는 쿼리 입니다.
- 여러 열을 기준으로 사용
 - `SELECT * FROM table_name ORDER BY 3, 1 DESC;`
 - 3번째 열을 기준으로 오름차순으로 정렬한 상태에서 1번째 열을 기준으로 내림차순으로 정렬하는 쿼리 입니다.

null 연산

- 아직 정의되지 않은 값으로 0 혹은 공백과는 다르다. (0은 숫자, 공백은 문자임)
- 테이블을 생성할 때 not null 또는 primary key로 정의되지 않으면 null을 포함할 수 있다. (때문에 create table을 할때 ~not null 이런식으로 자주 붙임)
- null값은 산술연산과 비교연산을 할 수 없다. 결과값은 무조건 null.
 - $1 + \text{null} = \text{null}$.
 - 컬럼 a가 1이고 컬럼 b가 null일때 연산결과는 null.
 - 하지만 null과 연산하고 싶을때 시스템에서 의미없는 문자로 바뀌어서 연산하는 경우가 많다. (이때 nvl 함수를 많이 쓴다)

null과 관련된 함수

1. nvl (표현식1, 표현식2) : null 처리 함수

- 오라클에서 쓰인다
- nvl(null 판단 대상, 'null일 때 대체값')을 말한다.
- 표현식 1 결과값이 null이면 표현식 2의 값을 준비.
- 아래 사진의 예제처럼 nvl('abc', 'xxx')

일 시엔 null이 아니면 abc가 적힌다.

그 아래 nvl(null, 'xxx')는 결과값이 null이므로

xxx가 적히게 된다.

ex: 인턴사원 아닌 사원들을 반환해줘 -> where nvl(deptcode, '인턴') != '인턴';

```
select
  nvl('abc', 'xxx'),
  nvl(null, 'xxx')
from
  dual;
```

2. isnull(표현식1, 표현식2) : isnull (null 판단 대상, 'null일 때 대체값')

- sql 서버에서 사용한다.
- nvl과 똑같은 기능을 수행(판단대상이 null이면 표현식2의 값이 옴)
- select isnull(null, 'nvl-ok') isnull_test
- 위의 결과값은 nvl-ok가 출력된다.

선수 테이블에서 포지션이 없는 경우 '없음' 으로 표시하는 예시

```
-- 오라클 (NVL)
SELECT PLAYER_NAME 선수명, NVL(POSITION, '없음') 포지션
FROM   PLAYER ;

-- sql server(ISNULL)
SELECT PLAYER_NAME 선수명, ISNULL(POSITION, '없음') 포지션
FROM   PLAYER ;
```


nvl과 isnull을 case로 표현해보는 것

NVL과 ISNULL은 CASE로 표현하는 것도 가능하다.

```
SELECT PLAYER_NAME 선수명,  
       CASE WHEN POSITION IS NULL  
            THEN '없음'  
            ELSE POSITION  
       END AS 포지션  
FROM   PLAYER ;
```

nvl로 실급여 계산

```
select
  emp_name,
  salary,
  bonus,
  nvl(bonus, 0),
  salary * 12,
  salary + (salary * nvl(bonus, 0)) as "실급여",
  '안녕'
from
  employee;
```

<- 실급여 계산

보너스를 받을때 식: 월급 +
(월급x보너스)

연봉 식: 월급x12 (1년 12개월)

결과값->

질의 결과 x						
SQL 인출된 모든 행: 24(0.005초)						
EMP_NAME	SALARY	BONUS	NVL(BONUS,0)	SALARY*12	실급여	'안녕'
1 선동일	8000000	0.3	0.3	96000000	10400000	안녕
2 송종기	6000000	(null)	0	72000000	6000000	안녕
3 노웅철	3700000	(null)	0	44400000	3700000	안녕
4 송은희	2800000	(null)	0	33600000	2800000	안녕
5 유재식	3400000	0.2	0.2	40800000	4080000	안녕
6 정중하	3900000	(null)	0	46800000	3900000	안녕
7 박나라	1800000	(null)	0	21600000	1800000	안녕
8 하이유	2200000	0.1	0.1	26400000	2420000	안녕
9 기채수	2500000	(null)	0	30000000	2500000	안녕

3. nullif(표현식1, 표현식2)

- sql server 함수(isnull처럼)
- 표현식1이 표현식2와 같으면 null, 아니면 표현식2를 리턴.
- 특정값을 null로 바꿀때 유용(ex: 한글hwp의 찾아 바꾸기 기능과 동일한 함수)

```
-- NULLIF (표현식1, 표현식2) : 표현식1 과 2과 같으면 NULL, 다르면 표현식 1 리턴  
  
-- MGR 7698 이면 NULL로 표시한다.  
SELECT ENAME, EMPNO, MGR, NULLIF(MGR, 7698) NUIF  
FROM EMP ;
```

4. coalesce(표현식1, 표현식2...)

- sql server 함수
- 표현식-컬럼명.
- 임의의 개수 표현식에서 null이 아닌 최초값을 결과로 리턴

```
SELECT  E_PLAYER_NAME,  
        ISNULL(E_PLAYER_NAME, 'HELLOWORLD') ISNULL값,      /*E_PLAYER_NAME이 널이면 HELLOWORLD*/  
        TEAM_ID,  
        NULLIF(Team_ID, 'K10') K10이면NULL,                /*TEAM_ID가 K10이면 NULL*/  
        E_PLAYER_NAME,  
        NICKNAME,  
        COALESCE(E_PLAYER_NAME, NICKNAME) 널아닌값          /*널 아닌 널아니면 최초값 출력*/  
FROM    PLAYER
```

E_PLAYER_NAME	NICKNAME	널아닌값
NULL	NULL	NULL
YOU, DONGWOO	NULL	YOU, DONGWOO
NULL	NICKNAME	NICKNAME
YOU, DONGWOO	NULL	YOU, DONGWOO
JEON, GIHYUN		JEON, GIHYUN

별칭 붙이기





- 결과집합의 컬럼명으로 사용됨
- **as** “별칭” - 이때 **as**와 “”(큰따옴표)는 생략 가능
- 숫자로 시작하는 별칭과 공백. 특수문자가 포함된 별칭은 꼭 큰따옴표(“”)로 감싸야한다.

```
select
  emp_name as "사원명",
  emp_no "주민번호", -- as 생략
  phone 전화번호, -- as, "" 생략
  dept_code "직급 코드", -- 띄어쓰기 있어서 ""로 감쌘
  job_code "1_직급" -- 숫자로 시작해서 ""로 감쌘
from
  employee;
```

distinct - 중복값 제거. **select** 구문 절에서 한번만 사용 가능. 여러 컬럼에 사용하면 여러컬럼값의 총합의 중복을 판단함.

```
select distinct
    dept_code
from
    employee
order by
    dept_code;
```

질의 결과 x

    SQL

DEPT_CODE

1 D1

2 D2

3 D5

4 D6

5 D8

6 D9

7 (null)

(문자열) 숫자연산자

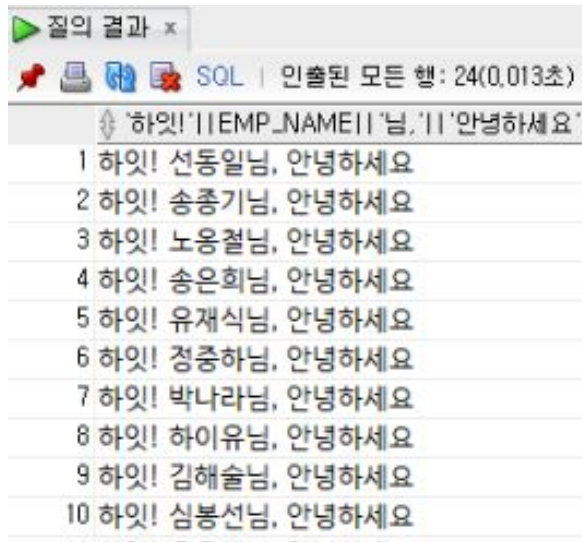
|| - 자바에선 **or**로 쓰이지만 **sql**에선 자바의 **+** 와 똑같다.

+ - **+(더하기)**는 숫자에서만 쓸수 있다(연산식)

문자인 경우에도 숫자로 자동형변환한다.

```
select
    '하잇: ' || emp_name || '님, ' || '안녕하세요'

from
    employee;
```



질의 결과 x

SQL | 인출된 모든 행: 24(0.013초)

	'하잇' EMP_NAME '님, ' '안녕하세요'
1	하잇! 선동일님, 안녕하세요
2	하잇! 송종기님, 안녕하세요
3	하잇! 노웅철님, 안녕하세요
4	하잇! 송은희님, 안녕하세요
5	하잇! 유재식님, 안녕하세요
6	하잇! 정중하님, 안녕하세요
7	하잇! 박나라님, 안녕하세요
8	하잇! 하이유님, 안녕하세요
9	하잇! 김해술님, 안녕하세요
10	하잇! 심봉선님, 안녕하세요

where

1. `=` : 같다
2. `!=` , `<>` , `^=` : 같지 않다
3. `>` : 크다 / `>=` : 크거나 같다 / `<` : 작다 / `<=` : 작거나 같다.
4. `between a and b` : `a` 이상 `b`이하
5. `like`, `not like` : 특정 문자 포함 데이터 확인 혹은 제외하고 출력 가능
6. `in`, `not in` : 값이 포함되거나 포함되지 않는 데이터를 추출하고 싶을 때 사용
7. `and` : 좌, 우항의 논리값이 모두 **true**라면 **true** 반환. 아니면 **false**
8. `or` : 좌, 우항의 논리값에서 둘 중 하나라도 **true**면 **true**반환. 그밖은 전부 **false**
9. `not` : 아닌 것을 찾을때 사용. (주로 반전될때: 1학년 말고 전부 찾아줘 같은 경우)

Between 연산자

between a and b : a(작은값) 이상 b(큰값) 이하 범위에 포함되면 true

- 숫자형과 날짜형 처리 가능. 'ex: 300만원 이상~500만원 이하, '20/01/01 ~ 22/01/01 사이에 새로 입사한 사람을 찾아줘 등등)
- 날짜형의 경우 형식은 yyyy/mm/dd, yy/mm/dd, yyyy-mm-dd, yymmdd 등 날짜포맷 문자열을 사용하면 알아서 날짜형으로 변환됨

LIKE, NOT LIKE

- 비교하는 값이 지정한 패턴을 만족하는 경우 **true** 반환
- **like**와 **not like**에선 %를 사용하거나 _를 사용하는 경우가 있음.
-
- %의 경우 0개 이상의 문자 결과가 나옴. (ex: 윤% : 윤씨 성인 사람을 다 찾아달라. 이름에 윤이 들어간 사람을 찾아야 하는 경우 %윤%)
- _의 경우 _ 1개가 글자 1글자. (ex: 윤__ 이렇게 하면 윤씨 성을 가진 이름 3글자인 사람이 나옴. 외자, 4글자 이상의 이름x)

+) escape 문자

이메일에서 @ 앞 _인 사원을 조회할때

where 조건절에서 email like '_____%' escape '\'; 하고 적기

LIKE, NOT LIKE

좀 더 자세한 내용 참고 포스팅: <https://kkh0977.tistory.com/1255>

```
/*
[LIKE , NOT LIKE 사용해 특정 문자 포함 데이터 확인 및 제외하고 출력 실시]
1. LIKE : 특정 문자를 포함하는 데이터를 출력합니다
2. NOT LIKE : 특정 문자를 포함하지 않는 데이터를 출력합니다
3. 컬럼 LIKE '문자%' : 특정 문자로 시작하는 데이터 확인
4. 컬럼 LIKE '%문자' : 특정 문자로 종료하는 데이터 확인
5. 컬럼 LIKE '%문자%' : 문자 시작 ~ 종료까지 특정 문자 포함 여부 확인
*/

-- [NOT LIKE 수행 실시]
SELECT T_NAME, T_DEPT
      FROM TEST_USER
WHERE  T_DEPT NOT LIKE '%백%' -- 백제, 후백제 필터링
      AND T_DEPT NOT LIKE '%고%' -- 고구려, 후고구려 필터링
      AND T_DEPT NOT LIKE '%신%' -- 신라, 통일신라 필터링
      AND T_DEPT NOT LIKE '%조%' -- 조선 필터링
ORDER BY T_DEPT DESC;
```

in : in연산 괄호 안에 쓴 목록의 값에 포함되어있으면 반환. **null**값은 포함x

ex: 부서코드 **in (F1, F2)**라고 하면 **F1**와 **F2** 에 포함된 사람들을 반환해달라는 **OR**의 뜻과 동일함. **F1**이나 **F2** 부서의 사람들을 알려줘. 하는거나 다름없음

not in : 위 **in**의 정반대. **not in (F1, F2)**라고 하면 **F1**이나 **F2**가 아닌 사람들 모두를 반환해줘 라는 뜻이 됨. (**in**과 차이가 있다면 **in**은 **or**의 뉘앙스지만 **not in**은 **and**.. 모든 대상이 아닌 것을 가리킴)

함수(FUNCTION)

- 유형 2가지
 1. 단일행 함수
 2. 그룹 함수

- 단일행 함수의 유형 5가지
 1. 문자처리
 2. 숫자처리
 3. 날짜처리
 4. (데이터) 형변환 - 묵시적/명시적 데이터형 변환
 5. 기타 함수

단일행 함수

1. 문자 함수

- `length` : 글자수(문자열 길이)를 반환함. 공백 포함해서 길이 반환.
- `lengthb` : 글자(문자열) 바이트수를 반환함. 영문자(1바이트), 공백(1바이트), 한글(3바이트)
- `instr`(문자열, 비교하고자 하는 값, 시작할 위치, 몇번째 비교값인지) : 찾고자 하는 문자열이 전체 문자열에서 몇번째 위치에 있는지 위치를 알려준다
(결과값: 위치번호)

`instr('hello world', 'e', 1, 1)` -> `hello world`라는 문자열에서 **e**를 찾고 싶은데 이걸 첫번째 글자부터 찾을지, 그리고 마지막은 문자열 내에 비교하고자 하는 값이 여러개 있는데 그 중 몇번째 비교값을 말하는 건지이다. 이때 첫번째 **e**를 말하는 것이다.

INSTR

- 읽어보면 좋을 참고 주소: <https://webstudynote.tistory.com/50> (제거함수에 대해서도 서술되어있다)

2) INSTR

- 형식 : **INSTR('문자열', '비교하고자 하는 값', 시작할 위치, 몇 번째 비교값인지)**
- 기능 : 내가 찾고자 하는 **문자열**이 전체 문자열에서 **몇번째 위치에 있는지** 위치를 알려준다.
- INSTR의 **결과값은 위치번호**이다.
- ex) INSTR('Hello, world!', 'e', 1, 1) ==> 2
- 함수명 유래 : 문자열에서 몇 번째 위치인지 (In string)에서 INSTR이 된 것 같다.
- Java와 비교 : Java의 index, last index of 와 유사하다.
- 마지막 argument(몇 번째 비교값인지)는 생략 가능하다. 생략 시, 비교값 중 첫번째 것을 이용한다. (디폴트:1)
ex) INSTR('Hello, world!', 'l', 1) ==> 3
- 마지막 argument는 문자열 내에 비교하고자 하는 값이 여러개가 있을 때, 그 중 몇번째 비교값을 말하는건지 알려준다.
ex) INSTR('Hello, world!', 'l', 1, 3) ==> 11 (3번째 l의 위치를 묻는 것.)
- 시작위치가 1이 아닐 경우, 시작위치 전의 문자는 비교를 하지 않는다. 번호는 시작위치가 1일 때와 동일하다.
ex) INSTR('Hello, world!', 'o', 1, 1) ==> 5
ex) INSTR('Hello, world!', 'o', 5, 1) ==> 5
ex) INSTR('Hello, world!', 'o', 7, 2) ==> 0 (7번째부터 검색하기 때문에 2번째 o가 없다고 생각함.)
ex) INSTR('Hello, world!', 'o', 7, 1) ==> 9

- **substr**(문자열, 시작위치, 길이) : 자르기 시작하는 위치와 잘랐을 때의 길이를 지정한다.
 - 자바의 **substring**과 다름없음. 마지막 길이는 생략가능하다. 생략하면 시작위치부터 문자열 끝까지 갖고 옴
 - **substr**('hello world!', 8) -> world!
 - **substr**('hello world', 3, 3) -> llo
 - **ex**: 사원 이메일 @앞을 조회하기
 1. **substr**(email, 1, **instr**(email, '@') -1) -> 이메일에서 골뱅이가 있는 위치번호를 **instr**로 먼저 찾고, 그 다음 **substr**로 이메일 전체에서 첫번째 글자~골뱅이 (골뱅이를 제외해야하므로 -1)전까지라는 뜻
 2. **substr**(email, 1, **length**(email) - **length**('@naver.com')) -> 이메일 전체 길이에서 이메일 @~ 의 길이값을 제외하면 아이디부분만 남음. 전체 이메일에서 첫번째 글자, 그리고 골뱅이 뒤의 길이값을 제외한 걸 넣으면 @앞을 알 수 있음.

- **lpad**(문자열, 길이, 채울 문자열) - 문자열 전체 길이에서 붙여넣을 문자열 길이를 뺀 만큼 남은 부분(채울 문자열)을 왼쪽에서부터 채움.
- **rpad**(문자열, 길이, 채울 문자열)- 문자열 전체 길이에서 붙여넣을 문자열 길이를 뺀 만큼 남은 부분(채울 문자열)을 오른쪽에서부터 채움.

```
select  
    lpad('123', 5, '0'),  
    lpad('12', 5, '0')  
from  
    dual;
```

질의 결과 x	
SQL 인출된 모든 행: 1(0.013초)	
LPAD('123',5,'0')	LPAD('12',5,'0')
1 00123	00012

숫자 함수

`abs()` : 절대값을 반환 (음수를 정수로 반환)

`mod()` : 나머지를 반환

`ceil()` : 소수점 기준 반올림. (ex: `ceil(234.56 * 10) / 10` -> 234.6)

`round()` : 소수점 기준 반올림. (ex: `round(234.56, 1)` -> 234.6)

`floor()` : 소수점 기준 버림 (ex: `floor(234.56 * 10) / 10` -> 234.5)

`trunc()` : 소수점 기준 버림(ex: `trunc(234.56, 1)` -> 234.5)

2. 날짜처리함수

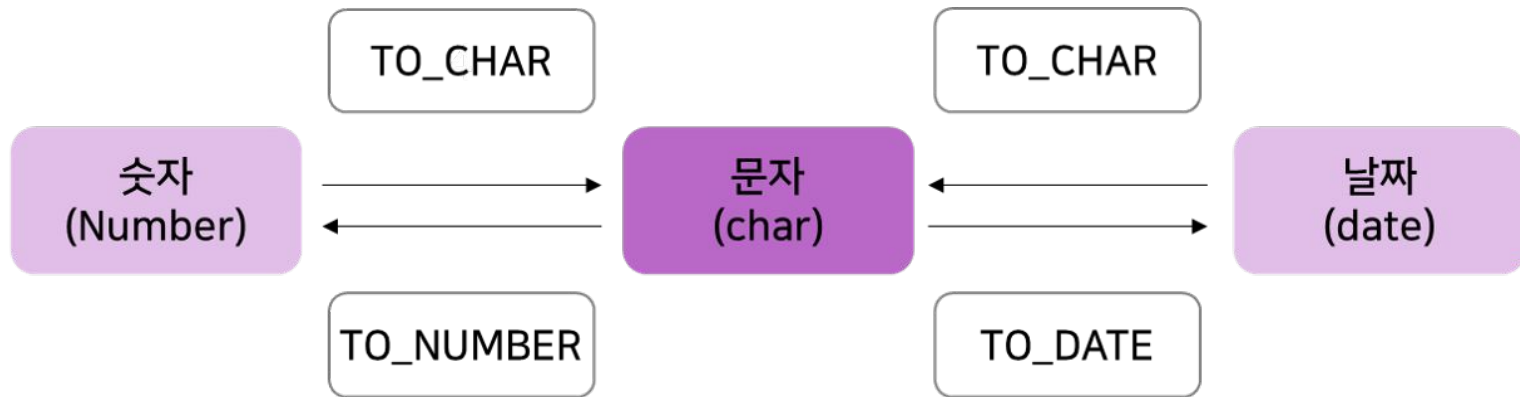
`add_months(date, number)` - 해당 날짜에 지정한 개월수를 더하거나 뺀 날짜형 반환.
31일이나 30일처럼 말일로 계산하면 해당 달의 말일을 반환.

`extract(from date, timestamp)` - 날짜정보 해당단위만 숫자형으로 반환. 시분초는 `date`가 아닌 `timestamp` 타입에서만 추출가능 (ex: year, month, day, hour, minute, second 다 가능. 단 hour의 경우 `timestamp`로 형변환 필요. `extract(hour from cast(sysdate as timestamp)) hh`)

`trunc(date)` - 날짜형에서 시분초 정보 제거 (ex: `to_char(sysdate, 'yyyy-mm-dd hh24:mi:ss')` "date")

`months_between(미래날짜, 과거날짜)` - 두 날짜의 개월수 차이를 반환 (ex: `months_between('23/01/01', sysdate)` 여기서 소수점이 나오면 `round` 문을 이용해서 반올림을 하면 깔끔하다

3. (데이터) 형변환 함수



※ Design by 쉰, <https://webstudynote.tistory.com/>

형변환 함수는 말 그대로 데이터를 변환해주는 함수.

to_char, to_date, to_yinterval, to_dsinterval 등이 있다.

to_char

1. date를 지정한 포맷형식으로 변환한 문자열 반환(날짜형>문자형으로)

day: 요일, dy: 짧은 요일, d: 숫자요일(월~일), am/pm:오전 오후

```
select
  to_char(sysdate, 'yyyy/mm/dd (dy) am hh24:mi:ss') 짧은요일,
  to_char(sysdate, 'yyyy/mm/dd (dy) am hh24:mi:ss', 'nls_date_language = korean') 요일을한국어로,
  to_char(sysdate, 'yyyy"년" mm"월" dd"일"') yyyy년mm월dd일,
  to_char(sysdate, 'fmyyyymmdd'), -- 포맷팅에 의해 생겨난 공백, 0등을 제거. 맨앞에 한번만 사용
  to_char(sysdate, 'dy', 'nls_date_language=korean') 요일
from
  dual;
```

짧은요일	요일을한국어로	YYYY년MM월DD일	TO_CHAR(SYSDATE,'FMYYYYMMDD')	요일
1 2022/04/15 (fri) AM 06:23:58	2022/04/15 (금) 오전 06:23:58	2022년 04월 15일	2022415	금

2. to_char(number, format): number 숫자형을 포맷형식으로 변환해서 문자형으로 반환 (숫자형>문자형으로) / fm으로 해당자리수가 없을 때 0 제거가능.

```
select
    to_char(123456789, 'fm9,999,999,999'),
    to_char(123.456, 'fm99999.99999'), -- 해당자리수가 없을때 소수점이상은 공백, 소수점미하는 0으로 처리. fm으로 제거가능
    to_char(123.456, 'fm00000.00000'), -- 해당자리수가 없을때 소수점이상/미하는 모두 0으로 처리. fm으로 제거불가
    to_char(123456789, 'FML9,999,999,999')
from
    dual;
```

질의 결과 x

SQL | 인출된 모든 행: 1(0.002초)

TO_CHAR(123456789, 'FM9,999,999,999')	TO_CHAR(123,456, 'FM99999,99,999')	TO_CHAR(123,456, 'FM00000,00000')	TO_CHAR(123456789, 'FML9,999,999,999')
123,456,789	123.456	00123.45600	₩123,456,789

to_number

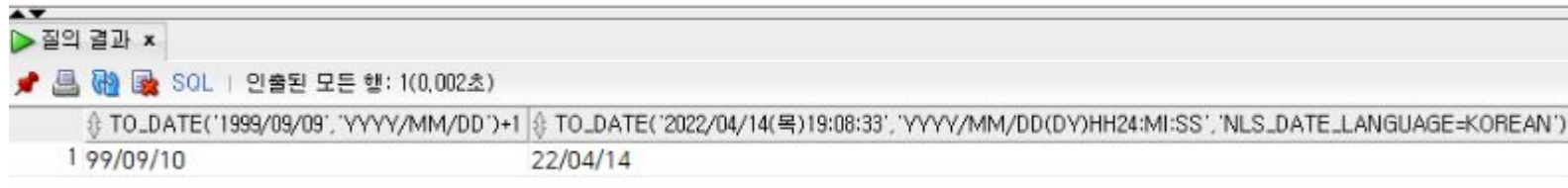
```
select
    to_number('₩123,456,789', 'L999,999,999') + 1,
    '1000' + 1
from
    dual;
```

질의 결과 x		
SQL 인출된 모든 행: 1(0,002초)		
TO_NUMBER('₩123,456,789','L999,999,999')+1	'1000'+1	
1	123456790	1001

to_date

- 날짜데이터인 문자열을 지정된 형식에 맞게 형변환 후 반환

```
select
    to_date('1999/09/09', 'yyyy/mm/dd') + 1,
    to_date('2022/04/14 (목) 19:08:33', 'yyyy/mm/dd (dy) hh24:mi:ss', 'nls_date_language = korean')
from
    dual;
```



실행 결과 x

SQL | 인출된 모든 행: 1(0.002초)

TO_DATE('1999/09/09','YYYY/MM/DD')+1	TO_DATE('2022/04/14(목)19:08:33','YYYY/MM/DD(DY)HH24:MI:SS','NLS_DATE_LANGUAGE=KOREAN')
1 99/09/10	22/04/14

to_ymininterval, to_dsinterval

1. to_ymininterval: 년 월을 더해주는 함수. (ex: 1년 2개월)
-> to_ymininterval('01-02') 1년 2개월을 더하는 식.
 2. to_dsinterval: 일, 시간 분 초를 더해주는 함수 (1일 2시간 1분 1초)
-> to_dsinterval('01 02:02:02') : 1일 2시간 2분 2초를 더하게 됨
- + 별개로 날짜정보추출로 extract 함수도 존재. (ex: extract(연월일시분초 from systimestamp))
 - + 참고 주소:
<https://blog.naver.com/PostView.nhn?blogId=regenesi90&logNo=222226229952>

기타함수

- 처리함수: nvl, nvl2 (nvl에 대한 내용은 위에 있기에 생략)
- 선택함수: decode, case 표현식 (when 값1(or 조건식1) then 결과값1 [else 기본값] end

decode(표현식, 값1, 결과값1, 값2, 결과값2, 값3, 결과값3.... [기본값])

-> select emp_name, decode(job_code, 'J1', '대표', 'J2', '부사장', 'J3', '부장', 'J4', '차장', 사원) job_name from employee;

case문

만약 case문을 이용해서 사원 주민등록번호를 보고 남여를 결과값이 출력되게 하고싶다면,

```
select
  emp_name,
  case substr(emp_no, 8, 1)
    when '1' then '남'
    when '2' then '여'
    when '3' then '남'
    when '4' then '여'
  end gender,
  case substr(emp_no, 8, 1)
    when '2' then '여'
    when '4' then '여'
    else '남'
  end gender
from
  employee;
```

substr는 문자열의 일부를 리턴. 주민번호(생년월일)는 8자리므로 생년월일 다음 첫숫자(8다음의 1)가 1,3이면 남자고 2,4면 여자. 두번째는 결과값은 똑같지만 **else**를 이용해 좀 더 간단해졌다.

그룹 함수

전체행을 하나의 그룹으로 처리해서 그룹당 하나의 결과를 반환

`group by`절을 통해 그룹처리 가능.

컬럼값이 `null`인 경우 반환x

`sum()` : 해당 열값을 모두 더한 결과값 리턴

`avg()` : 해당 열의 평균값을 리턴

`count()` : `null`값을 제외한 컬럼의 수를 반환

`max()` / `min()` : 가장 큰 값, 작은 값. (어디에 쓰이냐에 따라 조금씩 달라진다. 가장 최근과 과거가 될수도 있음)

group by

- 테이블 전체 행을 특정 컬럼이 동일한 행끼리 그룹핑 처리
- **group by**는 테이블 모든 행이 하나의 그룹처리로 보이는 상황에서 세부적으로 잡아낼 수 있다.
- **group by**는 null도 그룹처리를 하며, 가상컬럼을 갖고도 **group by**가 가능
- 또한 컬럼 순서가 중요하지 않다.
- **distinct**와 같이 두 컬럼 간 값이 동일한 행을 그룹핑 (**ex**: 컬럼1과 컬럼2의 **count** 수가 똑같다면 그걸 나란히 그룹핑)
- **distinct**와 **group by** 차이점에 대해: <http://www.gurubee.net/lecture/1032>

having

- group by가 와야 having을 쓸수 있음. 그룹핑이 되지 않은 상태에서 having 조건절이 올 수는 없다.
- 왜냐면 having절은 집계함수를 갖고 조건비교를 할 때 사용되기 때문

아래 예제는 사원수가 다섯 명이 넘는 부서와 사원수를 조회하는 예제이다.

```
1  SELECT b.dname, COUNT(a.empno) "사원수"
2  FROM emp a, dept b
3  WHERE a.deptno = b.deptno
4  GROUP BY dname
5  HAVING COUNT(a.empno) > 5;
```

DNAME	사원 수
SALES	6

group by, having 절의 참고 포스트. where과 having 절의 차이도 알수있음

<http://wiki.gurubee.net/pages/viewpage.action?pagelId=26743892>

아래 예제는 전체 월급이 5000을 초과하는 JOB에 대해서 JOB과 월급여 합계를 조회하는 예이다. 단 판매원(SALES)은 제외하고 월 급여 합계로 내림차순 정렬하였다.

```
1  SELECT job, SUM(sal) "급여합계"
2      FROM emp
3  WHERE job != 'SALES'      -- 판매원은 제외
4  GROUP BY job              -- 업무별로 Group By
5  HAVING SUM(sal) > 5000     -- 전체 월급이 5000을 초과하는
6  ORDER BY SUM(sal) DESC;   -- 월급여 합계로 내림차순 정렬
```

JOB	급여합계
MANAGER	8275
ANALYST	6000
SALESMAN	5600

ROLLUP / CUBE / GROUPING SETS / GROUPING

- 둘다 **group by** 조건절과 같이 쓰이는 함수.
- **roll up**은 그룹핑한걸 최종집계하기 위해 발달한 함수(**ex:** 남자 몇명, 여자 몇명 이렇게 분류 전체 모집 개수를 보여줌)로 소그룹간의 합계를 계산하는 함수.
- **rollup**의 경우 중간중간 소그룹 합계의 가상컬럼이 생성되며 빈곳은 **null**로 처리.
- **cube**는 항목들 간의 다차원적인 소계를 계산. **rollup** 함수와 달리 **group by**절에 명시한 모든 컬럼의 소그룹 합계를 계산함.
- **grouping**은 간접적으로 위의 집계 함수를 지원함
- **rollup/cube/grouping sets** 정리 참고 포스팅:
<https://for-my-wealthy-life.tistory.com/44>

ROLLUP

상품ID를 중심으로 그룹핑을 해서 소그룹 합계를 구하는 걸 볼 수 있음.

```
SELECT 상품ID, 월, SUM(매출액) AS 매출액  
FROM 월별매출  
GROUP BY ROLLUP(상품ID, 월);
```

	상품ID	월	매출액	
1	P001	2019.10	15000	
2	P001	2019.11	25000	
3	P001	(null)	40000	→ P001 합계
4	P002	2019.10	10000	
5	P002	2019.11	20000	
6	P002	(null)	30000	→ P002 합계
7	P003	2019.10	15000	
8	P003	2019.11	10000	
9	P003	(null)	25000	→ P003 합계
10	(null)	(null)	95000	→ 전체 합계

CUBE

상품ID별+월별+전체합계로 나오는걸 확인가능.

CUBE함수는 항목들 간의 다차원적인 소계를 계산합니다. ROLLUP과 달리 GROUP BY절에 명시한 모든 컬럼에 대해 소그룹 합계를 계산해 줍니다.

```
SELECT 상품ID, 월, SUM(매출액) AS 매출액
FROM 월별매출
GROUP BY CUBE(상품ID, 월);
```

↕	상품ID	↕	월	↕	매출액		
1	(null)		(null)		95000	→	전체 합계
2	(null)		2019.10		40000	→	2019.10 합계
3	(null)		2019.11		55000	→	2019.11 합계
4	P001		(null)		40000	→	P001 합계
5	P001		2019.10		15000		
6	P001		2019.11		25000		
7	P002		(null)		30000	→	P002 합계
8	P002		2019.10		10000		
9	P002		2019.11		20000		
10	P003		(null)		25000	→	P003 합계
11	P003		2019.10		15000		
12	P003		2019.11		10000		

GROUPING SETS

- ROLLUP과 CUBE에 비하면 간략.
- 각 소그룹별 합계만 보여줌

```
SELECT 상품ID, 월, SUM(매출액) AS 매출액  
FROM 월별매출  
GROUP BY GROUPING SETS(상품ID, 월);
```

상품ID	월	매출액
1 P001	(null)	40000
2 P003	(null)	25000
3 P002	(null)	30000
4 (null)	2019.11	55000
5 (null)	2019.10	40000

→ 상품별 합계

→ 월별 합계

GROUPING

```
SELECT
    CASE GROUPING(상품 ID) WHEN 1 THEN '모든 상품 ID' ELSE 상품 ID END AS 상품 ID,
    CASE GROUPING(월) WHEN 1 THEN '모든 월' ELSE 월 END AS 월,
    SUM(매출액) AS 매출액
FROM 월별매출
GROUP BY ROLLUP(상품 ID, 월);
```

	⇄ 상품ID	⇄ 월	⇄ 매출액
1	P001	2019.10	15000
2	P001	2019.11	25000
3	P001	모든 월	40000
4	P002	2019.10	10000
5	P002	2019.11	20000
6	P002	모든 월	30000
7	P003	2019.10	15000
8	P003	2019.11	10000
9	P003	모든 월	25000
10	모든 상품ID	모든 월	95000

맨 처음 단순 rollup 함수를 썼을 때 null 값이 나오던 게 전부 채워진 걸 확인 가능.

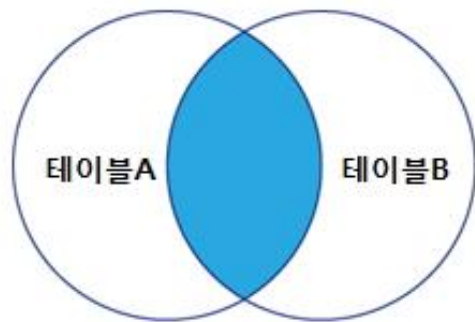
집계된 계산 결과에 대해 null 이면 1을 반환, 숫자가 채워져 있으면 0. 따라서 null 값일 때 모든 상품 id, 모든 월을 채워 넣어 달란 case when 구문

JOIN

- 두개의 테이블을 서로 묶어 하나의 결과를 만들어내는 것
- 일반적인 경우 행들은 **PK(기본키 PRIMARY KEY)**나 **FK(FOREIGN KEY)**값의 연관에 의해 **JOIN** 성립. 하지만 이런 **PK, FK** 관계없어도 논리적인 값들의 연관만으로 **JOIN** 성립이 가능.
- **from**절에 다수의 테이블이 나열되어있어도 2개 테이블이 먼저 조인, 합쳐진 조인과 남은 테이블이 조인되는 식(다중조인).
- **SQL 기본문법: INNER, OUTER, CROSS, SELF JOIN**

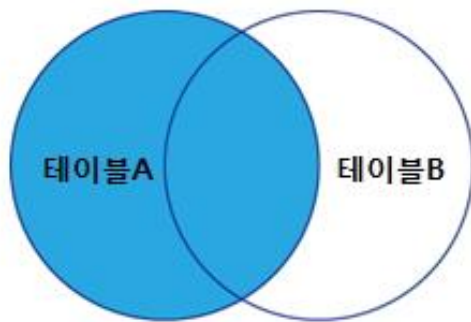
JOIN

• Join 종류



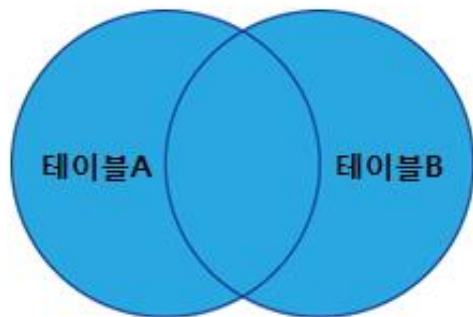
[INNER JOIN]

테이블A,B 모두 조건구문에 일치하는 데이터만 반환(교집합)



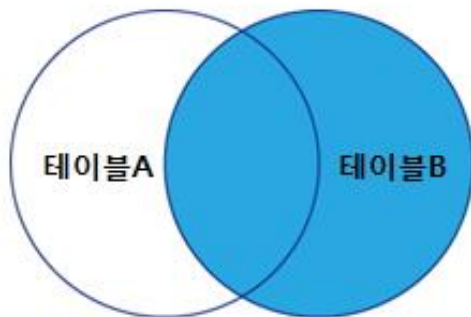
[LEFT OUTER JOIN]

테이블A 모두 반환
테이블B 조건구문에 일치하는 데이터만 반환



[FULL OUTER JOIN]

테이블 A,B 모두 반환



[RIGHT OUTER JOIN]

테이블B 모두 반환
테이블A 조건구문에 일치하는 데이터만 반환

INNER JOIN(=EQUI JOIN)

- 내부조인. 두 테이블간의 교집합을 의미.(특정 컬럼을 기준으로 정확히 매칭된 것만 추출)
- 각 테이블에서 기준컬럼이 null이거나 상대테이블에서 매칭되는게 없는 것 제외.
- inner join에서 inner는 생략 가능(default가 옵션으로 조인조건을 만족하는 행만 반환하기 때문)
- cross join, outer join과는 같이 사용 불가.
- 예제 food_a와 food_b의 이너조인을 해보기.

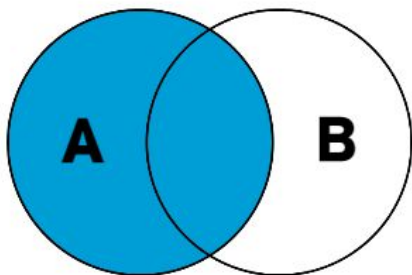
TABLE : FOOD_A			TABLE : FOOD_B	
ID	FOODNM		ID	FOODNM
1	돈까스		1	초밥
2	삼겹살		2	돈까스
3	초밥		3	칼국수
4	곱창전골		4	햄버거

식 : select <열 목록> from <첫번째 테이블> inner join <두번째 테이블> on <조인될 조건> [where 검색조건]

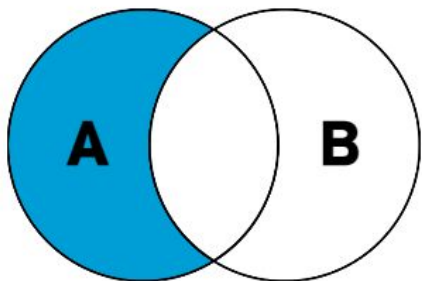
-> select * from food_a a inner join food_b b on a.foodnm = b.foodnm;

A.ID	A.FOODNM	B.ID	B.FOODNM
1	돈까스	2	돈까스
3	초밥	1	초밥

LEFT OUTER JOIN

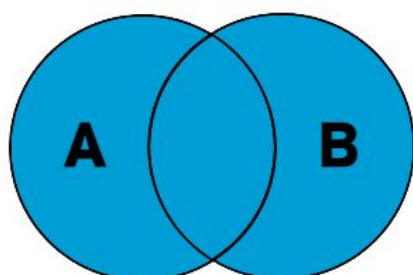


```
SELECT *  
FROM A a  
LEFT JOIN B b  
ON a.KEY = b.KEY
```

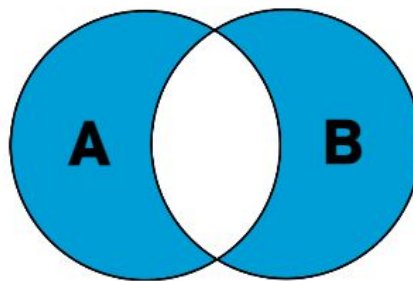


```
SELECT *  
FROM A a  
LEFT JOIN B b  
ON a.KEY = b.KEY  
WHERE b.KEY IS NULL
```

FULL OUTER JOIN

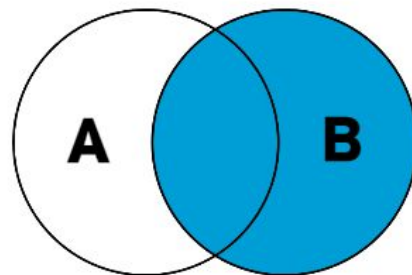


```
SELECT *  
FROM A a  
FULL OUTER JOIN B b  
ON a.KEY = b.KEY
```

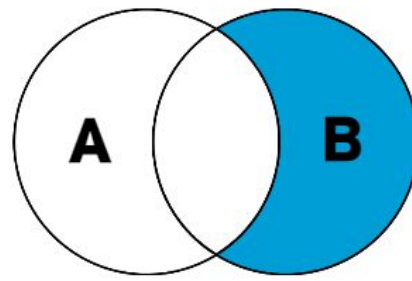


```
SELECT *  
FROM A a  
FULL OUTER JOIN B b  
ON a.KEY = b.KEY  
WHERE a.KEY IS NULL  
OR b.KEY IS NULL
```

RIGHT OUTER JOIN



```
SELECT *  
FROM A a  
RIGHT OUTER JOIN B b  
ON a.KEY = b.KEY
```



```
SELECT *  
FROM A a  
RIGHT OUTER JOIN B b  
ON a.KEY = b.KEY  
WHERE a.KEY IS NULL
```

OUTER 조인

1) LEFT OUTER JOIN

A LEFT OUTER JOIN B의 경우 왼쪽테이블 A는 전부 다 추출, B에선 A와 겹치는 행들만 추출된다. (A에 있는 B값만 갖고 오는 것임) A에 있지만 B에 없다면 NULL처리

예제) 아까의 FOOD_A와 FOOD_B에서 LEFT OUTER JOIN을 해보시오

```
SELECT * FROM_A A LEFT OUTER JOIN FOOD_B B ON A.FOODNM = B.FOODNM;
```

A.ID	A.FOODNM	B.ID	B.FOODNM
1	돈까스	2	돈까스
2	삼겹살	(NULL)	(NULL)
3	초밥	1	초밥
4	곱창전골	(NULL)	(NULL)

LEFT OUTER JOIN-(LEFT ONLY)

- A전체를 추출하지만 B와 매핑되는 것을 제외하고 추출하는것(A테이블-교집합)
- `SELECT * FROM FOOD_A A LEFT OUTER JOIN FOOD_B B ON A.FOODNM = B.FOODNM WHERE B.ID IS NULL;` (여기서 **where** 조건식을 사용한다. A만의 값을 찾아야 하는데 그 경우 B에서 없는 NULL일것)

A.ID	A.FOODNM	B.ID	B.FOODNM
2	삼겹살	(NULL)	(NULL)
4	곱창전골	(NULL)	(NULL)

RIGHT OUTER JOIN

A RIGHT OUTER JOIN B를 하면 B를 기준으로 A와 조인(LEFT OUTER JOIN와 정반대. RIGHT니까 오른쪽 B가 기준이 됨)

```
SELECT * FROM FOOD_A A RIGHT OUTER JOIN FOOD_B B ON A.FOODNM = B.FOODNM;
```

A.ID	A.FOODNM	B.ID	B.FOODNM
1	돈까스	2	돈까스
(NULL)	(NULL)	3	칼국수
3	초밥	1	초밥
(NULL)	(NULL)	4	햄버거

RIGHT OUTER JOIN - (RIGHT ONLY)

테이블 B에서 교집합을 뺀(A와 그룹핑으로 겹치는 행이 없는것) 순수 B의 값을 말한다.

```
SELECT * FROM FOOD_A A RIGHT OUTER JOIN FOOD_B B ON A.FOODNM =  
B.FOODNM WHERE A.ID_IS NULL;
```

A.ID	A.FOODNM	B.ID	B.FOODNM
(NULL)	(NULL)	3	칼국수
(NULL)	(NULL)	4	햄버거

FULL OUTER JOIN

INNER, LEFT OUTER, RIGHT OUTER 모든 조인 집합 출력 (합집합)

A, B 테이블 모든 데이터를 읽어 조인 결과를 생성함.

TABLE : FOOD_A		TABLE : FOOD_B	
ID	FOODNM	ID	FOODNM
1	돈까스	1	초밥
2	삼겹살	2	돈까스
3	초밥	3	칼국수
4	곱창전골	4	햄버거

<- 예제

위의 예제를 FULL OUTER JOIN 식으로 만들어보자면

```
SELECT * FROM FOOD_A A FULL OUTER JOIN FOOD_B B ON A.FOODNM = B.FOODNM;
```

아래의 결과처럼 FOOD_A, FOOD_B 양측 데이터를 합친게 나옴.

A.ID	A.FOODNM	B.ID	B.FOODNM
1	돈까스	2	돈까스
2	삼겹살	(NULL)	(NULL)
3	초밥	1	초밥
4	곱창전골	(NULL)	(NULL)
(NULL)	(NULL)	3	칼국수
(NULL)	(NULL)	4	햄버거

FULL OUTER JOIN - ONLY OUTER JOIN

FULL OUTER JOIN(합집합)에서 겹치는(교집합)을 제외한 각각에만 존재하는 데이터를 추출한것(합집합-교집합)

```
SELECT * FROM FOOD_A A FULL OUTER JOIN FOOD_B B ON A.FOODNM = B.FOODNM WHERE A.ID IS NULL AND B.ID IS NULL;
```

TABLE : FOOD_A		TABLE : FOOD_B	
ID	FOODNM	ID	FOODNM
1	돈까스	1	초밥
2	삼겹살	2	돈까스
3	초밥	3	칼국수
4	곱창전골	4	햄버거

NATURAL JOIN

- 반드시 두 테이블 간의 동일한 이름, 타입을 가진 컬럼이 필요하다.
- 조인에 이용되는 컬럼은 명시하지 않아도 자동으로 조인에 사용된다.
- 동일한 이름을 갖는 컬럼이 있어도 데이터 타입이 다르면 에러 발생.
- 조인하는 테이블간의 동일 컬럼이 **select** 절에 기술되어도 테이블 이름을 생략해야 한다.
- 네츨럴 조인은 잘 사용되지 않는 편이다. 이너 조인과 다르게 조인 컬럼을 명시하지 않아도 두 테이블 간 동일한 이름의 컬럼을 자동으로 찾아 조인하기 때문. 동일한 컬럼이 하나가 아닌 여러개라면 여러개 컬럼이 모두 동일한 값을 가졌다는 이유만으로 결과값으로 추출되기에 기대값과 다르거나 혹은 아예 추출되지 않을 수도 있다.

natural join과 using 조건문

- 네츄럴 조인의 문제점(테이블간 동일 이름과 형식 컬럼이 2개 이상인 경우 사용불가)이 있는 편이다. 동일한 이름과 형식의 컬럼을 2개 이상 사용할 경우 **using**절을 이용하면 조인문을 구사 가능하다. (단, **natural** 조인문에 **using**절은 함께 사용될 수 없다.)
- 기본 구조: **select** 컬럼명 **from** 테이블1 **join** 테이블 2 **using** 조인컬럼 **where** 조건절;
- 보면 좋을 참고 설명: <https://keep-cool.tistory.com/41>

ON / USING / WHERE의 차이

sql에서 join을 쓸때 on/using/where은 각각 아래와 같은 특징이 있다

- on: 다양한 join식 가능. 조건식 a.컬럼 = b.컬럼 (on은 join절보다 먼저 실행됨. on 조건으로 필터링된 컬럼들간의 join이 이뤄짐)
- using은 두 테이블 간 필드 이름이 동일한 경우 사용(이퀄조인에만 해당. 두 테이블이 동일 컬럼이어야함). 조인에 사용할 테이블, 뷰, 서브쿼리 중 1개 기술.
- where은 join을 한 후 where에서 필터링. (join 결과 후 where 조건으로 필터링)
- 차이점 참고 사이트: <https://developyo.tistory.com/121> ,
- on과 using 차이점 참고 사이트: <https://pakker.tistory.com/115>

CROSS JOIN

- 테이블 간 조인 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말함 (경우의 수).



위는 예제 테이블

`select * from T1 cross join T2;`

결과는 아래와 같음

NUMBER	TEXT
1	A
1	B
2	A
2	B
3	A
3	B

SELF JOIN

- 같은 테이블에서 특정 컬럼을 기준으로 매칭되는 결과를 출력하는 조인. 같은 테이블의 데이터를 각각 분류한 후 조인함.
- 같은 테이블(테이블명과 컬럼명 동일)이므로 테이블명은 무조건 별칭을 붙이는게 좋다.
- 아래와 같은 테이블이 있을때 **MNGID**(상위직책자의 아이디 컬럼)의 사람들도

TABLE : EMP		
ID	NAME	MNGID
1	Alice	
2	Benny	1
3	Cindy	2
4	David	1

결국 사원이기 때문에 emp테이블(전체 테이블)에 있을것
따라서 식은 `select * from emp e left outer join emp m
on e.mngid = m.id;` 가 되어야 함.

MULTIPLE JOIN

- 다중조인. 3개 이상 테이블을 동시에 조인하는 것. 공통 컬럼을 잘 찾아야함
- ansi 표준문법에선 조인되는 순서가 중요하다
- 여기서 ansi sql이란? dbms(오라클, my-sql, db2 등)에서 각기 다른 sql을 사용하므로 미국 표준협회에서 이걸 표준화하여 표준sql문을 정립해놓음
(간단하게 한국어에서도 사투리들이 되게 많아 표준어가 존재하는 것과 똑같다...)
- 테이블을 조인하는 순서에 따라 결과테이블 구성과 행 개수가 달라질 수 있음.
때문에 가장 첫번째 테이블로는 **select** 문에서 가장많은 열을 가져와야 할
테이블을 우선시 하는게 좋음.
- 다중조인을 할 때 시작을 **left join**으로 했다면 나머지도 **left join**으로 가야함.
- 다중조인에 대해선 아래의 사이트가 굉장히 잘 정리되어 있어 링크:

<https://kimsyoung.tistory.com/entry/3%E0%B0%9C-%EC%9D%B4%EC%83%81%EC%9D%98-%ED%85%8C%EC%9D%B4%EB%B8%94-LEFT-JOIN-%ED%95%98%E0%B8%B0>