1. **서브쿼리/SET/계층형**
2. **서브쿼리(SubQuery)**

실제 프로젝트 현장에서 접하게 되는 여러 SQL문장들 중에서 아주 유용하게 사용되는 문장들

중 하나이다.

서브쿼리란 하나의 SQL문장 내부에 존재하는 또 다른 SELECT문장을 말한다.

ex) 사원들의 월급 평균을 구했다면, 전체 사원들 중 평균 급여보다 낮은 급여를 받는

사원들의 명단을 추출해야 한다면 쿼리를 어떻게 작성해야 하나?

1) 평균 월급을 구한다.

2) 이 값보다 월급이 작은 사원들의 명단을 추출한다.

|  |
| --- |
| -- 1) 이 결과값으로 6462가 리턴되었다면  **SELECT** **ROUND(AVG**(salary))  **FROM** employees; |
| -- 2)  **SELECT** employee\_id, first\_name, last\_name  **FROM** employees  **WHERE** salary < 6462; |

Ex) 서브쿼리를 활용

|  |
| --- |
| **SELECT** employee\_id, first\_name, last\_name  **FROM** employees  **WHERE** salary < ( **SELECT** **ROUND(AVG**(salary))  **FROM** employees ); |

\*) 이전 쿼리와 동일한 결과를 리턴한다.

\*) 두 개의 쿼리는 데이터 연관성이 없이 독립적으로 실행되었다.

경우에 따라서는 데이터의 참조나 공유가 발생할 수도 있다.

* 1. **서브 쿼리의 구분**

|  |
| --- |
| 메인쿼리와의 연관성에 따라  1) 연관성 없는 서브쿼리(Noncorrelated Subquery)  2) 연관성 있는 서브쿼리(Correlated Subquery)  서브쿼리가 위치한 곳에 따라  1) 일반 서브쿼리  2) 인라인 뷰( inline view) : FROM절에 위치 => 무조건 연관성 없는 서브쿼리에 포함됨  3) 중첩쿼리(nested query) : WHERE절에 위치 |

[1] 연관성 없는 서브쿼리

서브쿼리와 메인쿼리와의 사이에 데이터의 연관성이 없는 서브쿼리를 말한다.

아래는 연관성 없는 서브쿼리의 일반적 형태

주가 없는 도시에 위치한 부서정보를 조회하는 쿼리임

서브쿼리 : STATE\_PROVINCE값이 NULL인 도시의 LOCATION\_ID값을 반환

메인쿼리 : 서브쿼리의 결과값에 포함되는 부서를 조회

|  |
| --- |
| **SELECT** \*  **FROM** departments  **WHERE** location\_id IN ( **SELECT** location\_id  **FROM** locations  **WHERE** state\_province IS NULL ); |

\*) 위의 쿼리를 메인쿼리의 입장에서 볼 때 처리되는 결과

|  |
| --- |
| **SELECT** \*  **FROM** departments  **WHERE** location\_id IN ( 1000, 1100, 1300, 2000, 2300, 2400); |

ex) 평균월급보다 적은 급여를 받는 사원들의 명단을 조회

|  |
| --- |
| **SELECT** employee\_id, first\_name, last\_name  **FROM** employees  **WHERE** salary < ( **SELECT ROUND(AVG**(salary))  **FROM** employees ); |

\*) 이전에 사용한 쿼리는 반환 row가 1개 이상이지만, 지금은 집계함수를 사용하여 단하나

의 로우만 반환된다.

반환하는 로우나 컬럼수에 따라서 서브쿼리는 다음과 같이 구별된다.

1) 단일 로우, 단일 컬럼을 반환하는 서브쿼리(평균값을 반환하는 두번째 쿼리)

2) 다중 로우, 단일 컬럼을 반환하는 서브쿼리(부서번호를 반환하는 첫번째 쿼리)

3) 다중 컬럼을 반환하는 서브쿼리

**1) 단일 로우, 단일 컬럼을 반환**

대부분 집계함수가 포함된 경우가 많음

Ex) 월급이 가장 많은 사원 정보 조회

|  |
| --- |
| **SELECT** a.first\_name, a.last\_name, b.job\_title  **FROM** employees a, jobs b  **WHERE** a.salary = ( **SELECT MAX**(salary)  **FROM** employees )  **AND** a.job\_id = b.job\_id; |

Ex) 월급이 가장 적은 사원 정보 조회

|  |
| --- |
| **SELECT** a.first\_name, a.last\_name, b.job\_title  **FROM** employees a, jobs b  **WHERE** a.salary = ( **SELECT** MIN(salary)  **FROM** employees)  **AND** a.job\_id = b.job\_id; |

Ex) 미국내에서 근무하는 사원들의 평균 월급보다 많은 급여를 받는 사원 명단 조회

|  |
| --- |
| **SELECT** a.first\_name, a.last\_name, b.job\_title  **FROM** employees a, jobs b  **WHERE** a.salary = ( **SELECT AVG**(emp.salary)  **FROM** employees emp,  departments dep,  locations loc  **WHERE** emp.department\_id = dep.department\_id  **AND** dep.location\_id = loc.location\_id  AND loc.country\_id = 'US' )  **AND** a.job\_id = b.job\_id; |

**-- 2) 다중 로우, 단일 컬럼을 반환**

일반적인 쿼리형태가 많다.

Ex) 미국에 위치한 부서를 알고 싶다.

|  |
| --- |
| **SELECT** dep.department\_name  **FROM** departments dep, locations loc  **WHERE** dep.location\_id = loc.location\_id  **AND** loc.country\_id = 'US'; |

Ex) 서브쿼리를 사용하여 위와 동일결과 리턴

서브쿼리가 다중 로우를 반환하므로 동등연산자(=)가 아닌 IN 연산자가 사용되었다.

|  |
| --- |
| **SELECT** department\_name  **FROM** departments  **WHERE** location\_id IN ( **SELECT** location\_id  **FROM** locations  **WHERE** country\_id = 'US'); |

Ex) 구매부서에 속한 사원들의 월급을 반환하는 쿼리이다.

|  |
| --- |
| **SELECT** salary  **FROM** employees  **WHERE** department\_id = 30; |

Ex) 만약 위의 쿼리를 수정하여 구매부서에 속한 사원들의 월급보다 높은 월급을 받는 사원명단을

추출하는 쿼리를 작성해야 한다면?

|  |
| --- |
| **SELECT** employee\_id, department\_id, salary  **FROM** employees  **WHERE** salary > **ANY** ( **SELECT** salary  **FROM** employees  **WHERE** department\_id = 30); |

Ex) 위와 동일 결과 리턴

|  |
| --- |
| **SELECT** employee\_id, salary  **FROM** employees  **WHERE** salary > ( **SELECT** MIN(salary)  **FROM** employees  **WHERE** department\_id = 30); |

Ex) 구매부서의 월급보다 많은 월급을 받는 사원들 조회

|  |
| --- |
| **SELECT** employee\_id, department\_id, salary  **FROM** employees  **WHERE** salary > **ALL** ( **SELECT** salary  **FROM** employees  **WHERE** department\_id = 30); |

Ex) 위와 동일 결과 리턴

|  |
| --- |
| **SELECT** employee\_id, department\_id, salary  **FROM** employees  **WHERE** salary > ( **SELECT** MAX(salary)  **FROM** employees  **WHERE** department\_id = 30); |

Ex) 서브쿼리의 여러 개의 결과 중 단 하나의 값만 같더라도 메인쿼리의 결과에 포함된다.

|  |
| --- |
| **SELECT** employee\_id, department\_id, salary  **FROM** employees  **WHERE** salary = **ANY** ( **SELECT** salary  FROM employees  WHERE department\_id = 30); |

Ex) 위와 동일한 연산

|  |
| --- |
| **SELECT** employee\_id, department\_id, salary  **FROM** employees  **WHERE** salary **IN ( SELECT** salary  **FROM** employees  **WHERE** department\_id = 30); |

\*) = ANY와 IN이 같다고 해서 그 반대의 경우도 같지는 않다.

즉 <> ANY 는 NOT IN 과 같지 않다.

- NOT IN : 서브쿼리의 결과에 포함되지 않는 것을 추출함

- <> ANY : 서브쿼리의 결과 중 단 하나의 값과 같지 않기만 한다면 조건을 만족하여

결과에 포함한다.

Ex) 부서번호가 10과 20이 아닌 모든 것을 추출한다.

|  |
| --- |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id NOT IN ( 10,20); |

Ex) 부서번호가 10이라면 20이 아니니깐 추출되고,

부서번호가 20이라면 10이 아니니깐 추출된다.

나머지는 위의 결과와 동일하다. 즉 위보다 2개가 더 추출된다.

|  |
| --- |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **<>** **ANY** ( 10,20 ); |

Ex) =ALL은 서브쿼리의 결과로 오는 모든 값과 같은 건을 추출한다.

WHERE 조건에 오는 메인쿼리의 컬럼값을 동시에 여러 값과 같음을 비교하는 것이므로 어떤 결과도 추출될 수 없다., 단 서브쿼리가 단 하나의 값만 반환될 때는 비교가 가능하다.

|  |
| --- |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **=** **ALL** ( **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **IN** (10, 20));  -- 결과값 0 row 발생 |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **= ALL** ( 10,20 );  -- 위와 동일 쿼리 : 결과값 0 row 발생 |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **= ALL** ( **SELECT** department\_id  **FROM** departments  **WHERE** department\_id = 10);  -- 결과� : 1 row 발생 |
| **SELECT** department\_id  **FROM** departments  **WHERE** department\_id **= ALL** (10);  -- 위와 동일 쿼리 : 결과값 1row 발생 |

-- 서브쿼리는 SELECT문장이 아닌 Update, DELETE문장에서도 사용 가능하다.

UPDATE : SET 절과 WHERE절에서 사용가능

DELETE : WHERE절에서 서브쿼리 사용가능

UPDATE문의 SET절에서는 컬럼값을 특정 값으로 갱신하기 때문에 반드시 단일 로우를 반환하는

서브쿼리가 와야 한다.

-- 실제로 다중 로우를 발생할것 같지만, 부서번호가 10인 것은 단 한건 존재하기 때문에 오류가 발생하지 않는다.

|  |
| --- |
| **UPDATE** employees  **SET** salary **= ( SELECT** salary  **FROM** employees  **WHERE** department\_id = 10 ); |

-- 다중 로우를 발생시키기 때문에 오류가 발생한다.

-- 동일한 형태의 서브쿼리가 사용되었더라도 실제 데이터에 따라 오류가 발생하기도 하고

정상적으로 수행되기도 한다.

|  |
| --- |
| **UPDATE** employees  **SET** salary **=** ( **SELECT** salary  **FROM** employees  **WHERE** department\_id **IN** (10,20) ); |

-- Update 문과 Delete문의 Where 절에서 서브쿼리가 사용된 문장들

|  |
| --- |
| . **UPDATE** employees  **SET** salary **= ( SELECT** salary  **FROM** employees  **WHERE** department\_id = 10)  **WHERE** department\_id **IN** ( **SELECT** department\_id  **FROM** departments  **WHERE** manager\_id **IS NOT NULL** ); |
| **DELETE** countries  **WHERE** country\_id **NOT IN** ( **SELECT** country\_id  **FROM** locations); |

**-- 3) 다중 컬럼을 반환**

서브쿼리가 하나 이상의 컬럼을 반환할 수도 있다.

서브쿼리도 SELECT 문장이기때문이다.

서브쿼리는 단일로우 다중컬럼을 반환하거나 다중로우 다중컬럼을 반환할 수 있다.

-- 이 회사의 사장이 속한 부서의 정보를 추출하는 쿼리문

SELECT department\_id, department\_name

FROM departments

WHERE ( department\_id, manager\_id ) IN ( SELECT department\_id, employee\_id

FROM employees

WHERE manager\_id IS NULL );

-- example) 월별로 부서별 사원들의 급여 정보를 저장할 MONTH\_SALARY 라는 테이블 생성

CREATE TABLE month\_salary (

magam\_date DATE NOT NULL, /\* 마감일 \*/

department\_id NUMBER, /\* 부서번호 \*/

emp\_count NUMBER, /\* 사원수 \*/

total\_salary NUMBER(9,2), /\* 급여총액 \*/

average\_salary NUMBER(9,2) /\* 급여평균 \*/

);

-- example) 매월 말일이 되면 EMPLOYEES 테이블에 있는 정보를 기초로 해서 부서별 사원수,

급여 총액, 급여 평균 데이터를 집계해서 위 테이블에 저장한다고 가정

예) 마감일과 부서 정보 저장

INSERT INTO month\_salary ( magam\_date, department\_id)

SELECT LAST\_DAY(SYSDATE),

department\_id

FROM employees

GROUP BY department\_id;

-- example) 결과 확인

SELECT magam\_date, department\_id

FROM month\_salary;

-- example) 부서별 급여 통계 정보를 갱신

사원수, 급여총액, 평균급여액을 갱신

UPDATE month\_salary a

SET a.emp\_count = ( SELECT COUNT(\*)

FROM employees k

WHERE k.department\_id = a.department\_id

GROUP BY k.department\_id),

a.total\_salary = ( SELECT SUM(k.salary)

FROM employees k

WHERE k.department\_id = a.department\_id

GROUP BY k.department\_id),

a.average\_salary = ( SELECT ROUND(SUM(k.salary))

FROM employees k

WHERE k.department\_id = a.department\_id

GROUP BY k.department\_id);

-- example) 위 문장을 두 개 이상의 컬럼을 반환하는 서브쿼리를 사용하면

훨씬 간단해지게 됨

UPDATE month\_salary a

SET ( a.emp\_count, a.total\_salary, a.average\_salary ) =

( SELECT COUNT(\*), SUM(k.salary), ROUND(SUM(k.salary))

FROM employees k

WHERE k.department\_id = a.department\_id

GROUP BY k.department\_id);

-- example) 결과물 출력

SELECT \*

FROM month\_salary;

**-- [2] 연관성 있는 서브쿼리**

연관성 없는 서브쿼리 : 메인쿼리와 별도로 서브쿼리가 먼저 수행된 후에 그 결과를

메인쿼리와 비교하는 형태임

연관성 있는 서브쿼리 : 서브쿼리에서 추출되는 혹은 추출될 데이터 집합들이 메인쿼리

에서 추출되는 데이터 집합들과 같거나 혹은 다르거나 하는

임의의 관계가 성립된다.

-- 연관성 없는 서브쿼리(IN 연산자 사용)

SELECT count(\*)

FROM employees

WHERE department\_id **IN** ( SELECT department\_id

FROM departments

WHERE manager\_id IS NOT NULL);

-- 연관성 있는 서브쿼리(EXISTS 연산자 사용)

SELECT count(\*)

FROM employees emp

WHERE **EXISTS** ( SELECT 1

FROM departments dep

WHERE dep.manager\_id IS NOT NULL

AND emp.department\_id = dep.department\_id);

-- 위의 두개의 쿼리는 같은 결과를 리턴한다.

데이터 연관성이 있다는 것은 서브쿼리와 메인쿼리 사이에 조인이 사용되었음을

의미한다.

-- EXSIST 연산자 : 서브쿼리가 반환하는 결과에 메인쿼리에서 추출될 데이터들이 존재하기만

한다면 조건을 만족함

-- 사원이름과 소속부서명을 가져오는 쿼리임

SELECT emp.first\_name || ' ' || emp.last\_name emp\_names,

emp.department\_id,

dep.department\_name

FROM employees emp, departments dep

WHERE emp.department\_id = dep.department\_id;

-- 위의 내용을 서브쿼리를 사용하여 동일하게 구현할 수 있다.

SELECT emp.first\_name || ' ' || emp.last\_name emp\_names,

emp.department\_id,

( SELECT dep.department\_name

FROM departments dep

WHERE dep.department\_id = emp.department\_id) dep\_name

FROM employees emp;

DELETE month\_salary a

WHERE EXISTS ( SELECT 1

FROM employees emp

WHERE emp.department\_id = a.department\_id

GROUP BY emp.department\_id

HAVING COUNT(\*) = a.emp\_count

AND SUM(emp.salary) = 51600 );

SELECT \*

FROM month\_salary;

**[3] 인라인 뷰**

SQL문장에서 FROM 절에 사용된 서브쿼리를 말한다.

SQL문장에서만 사용가능하며 데이터베이스 객체가 아니다.

SQL문장이 실행될 때 생성되어 실행 종료시에 사라지는 임시성 뷰라고 할 수 있다.

뷰 : 여러 테이블 간에 존재하는 컬럼과 데이터들을 하나로 모아서 테이블처럼

사용할 수 있다는 장점을 가지고 있다.

인라인 뷰는 연관성 없는 서브쿼리에 속한다.

- 평균 급여보다 높고 최대 급여보다 낮은 월급을 받는 사원 리스트 조회

-- 중요점은 평균 급여와 최대 급여인데, 연관성 없는 서브쿼리를 2개 사용하여 결과를

추출하였다.

만약, 이 결과에 평균 급여와 최대 급여도 보여줘야 한다면 어떻게 해야 할까?

평균 및 최대 급여액은 연관성 없는 서브쿼리로 구해왔기 때문에 SELECT리스트에 사용될

수 없다.

평균 및 최대 급여액을 구하는 서브쿼리를 인라인 뷰로 만든다면 서브쿼리를 단 1개만

사용하더라도 동일한 결과를 표현할 수 있다.

SELECT a.employee\_id, a.first\_name || ' ' || a.last\_name names, a.salary

FROM employees a

WHERE a.salary >= ( SELECT AVG(salary)

FROM employees )

AND a.salary <= ( SELECT MAX(salary)

FROM employees )

ORDER BY a.salary DESC;

-- 위와 동일결과

SELECT a.employee\_id, a.first\_name || ' ' || a.last\_name names, a.salary,

ROUND(b.avgs), b.maxs

FROM employees a,

( SELECT AVG(salary) avgs,

MAX(salary) maxs

FROM employees ) b

WHERE a.salary BETWEEN b.avgs AND b.maxs

ORDER BY a.salary DESC;

-- 입사월별 사원수 조회

-- 원하는 결과?

컬럼 12, 로우 12개가 나온다.

SELECT DECODE(TO\_CHAR(hire\_date,'mm'), '01', COUNT(\*), 0) "1월",

DECODE(TO\_CHAR(hire\_date,'mm'), '02', COUNT(\*), 0) "2월",

DECODE(TO\_CHAR(hire\_date,'mm'), '03', COUNT(\*), 0) "3월",

DECODE(TO\_CHAR(hire\_date,'mm'), '04', COUNT(\*), 0) "4월",

DECODE(TO\_CHAR(hire\_date,'mm'), '05', COUNT(\*), 0) "5월",

DECODE(TO\_CHAR(hire\_date,'mm'), '06', COUNT(\*), 0) "6월",

DECODE(TO\_CHAR(hire\_date,'mm'), '07', COUNT(\*), 0) "7월",

DECODE(TO\_CHAR(hire\_date,'mm'), '08', COUNT(\*), 0) "8월",

DECODE(TO\_CHAR(hire\_date,'mm'), '09', COUNT(\*), 0) "9월",

DECODE(TO\_CHAR(hire\_date,'mm'), '10', COUNT(\*), 0) "10월",

DECODE(TO\_CHAR(hire\_date,'mm'), '11', COUNT(\*), 0) "11월",

DECODE(TO\_CHAR(hire\_date,'mm'), '12', COUNT(\*), 0) "12월"

FROM employees

GROUP BY TO\_CHAR(hire\_date,'mm')

ORDER BY TO\_CHAR(hire\_date,'mm');

-- 위의 문제를 해결하였다.

SELECT SUM(m1) "1월", SUM(m2) "2월", SUM(m3) "3월",

SUM(m4) "4월", SUM(m5) "5월", SUM(m6) "6월",

SUM(m7) "7월", SUM(m8) "8월", SUM(m9) "9월",

SUM(m10) "10월", SUM(m11) "11월", SUM(m12) "12월"

FROM (

SELECT DECODE(TO\_CHAR(hire\_date,'mm'), '01', COUNT(\*), 0) m1,

DECODE(TO\_CHAR(hire\_date,'mm'), '02', COUNT(\*), 0) m2,

DECODE(TO\_CHAR(hire\_date,'mm'), '03', COUNT(\*), 0) m3,

DECODE(TO\_CHAR(hire\_date,'mm'), '04', COUNT(\*), 0) m4,

DECODE(TO\_CHAR(hire\_date,'mm'), '05', COUNT(\*), 0) m5,

DECODE(TO\_CHAR(hire\_date,'mm'), '06', COUNT(\*), 0) m6,

DECODE(TO\_CHAR(hire\_date,'mm'), '07', COUNT(\*), 0) m7,

DECODE(TO\_CHAR(hire\_date,'mm'), '08', COUNT(\*), 0) m8,

DECODE(TO\_CHAR(hire\_date,'mm'), '09', COUNT(\*), 0) m9,

DECODE(TO\_CHAR(hire\_date,'mm'), '10', COUNT(\*), 0) m10,

DECODE(TO\_CHAR(hire\_date,'mm'), '11', COUNT(\*), 0) m11,

DECODE(TO\_CHAR(hire\_date,'mm'), '12', COUNT(\*), 0) m12

FROM employees

GROUP BY TO\_CHAR(hire\_date,'mm')

);

-- example) 월급순으로 상위 10명의 명단을 추출해야 한다면?

-- (1) 월급순으로 내림차순한다.

SELECT employee\_id, first\_name, last\_name, salary

FROM employees

ORDER BY salary DESC;

-- (2) row의 갯수로 짤라낸다.

-- 먼저 10개의 로우를 추출하여 내림차순을 하기 때문에

원하는 결과가 발생되지 않는다.

SELECT employee\_id, first\_name, last\_name, salary

FROM employees

WHERE ROWNUM < 11

ORDER BY salary DESC;

--- 해결 방법

월급순으로 내림차순 정렬된 결과를 인라인뷰로 만들고

이 결과를 가지고10개의 로우를 잘라낸다.

SELECT \*

FROM ( SELECT employee\_id, first\_name, last\_name, salary

FROM employees

ORDER BY salary DESC )

WHERE ROWNUM < 11;

-- 가장 적은 월급을 받는 10명의 사원을 추출하려면?

SELECT \*

FROM ( SELECT employee\_id, first\_name, last\_name, salary

FROM employees

ORDER BY salary ASC )

WHERE ROWNUM < 11;

**2. SET 연산자**

서브쿼리 : 메인쿼리가 하나 혹은 그 이상의 SELECT 문장을 포함한 형태

서브쿼리는 메인쿼리에 종속된 SELECT 문장이며,

최종적으로 산출되는 결과는 서브쿼리가 아닌 메인쿼리에 의해 좌우된다.

SET : 여러개의 SELECT 문장을 하나로 취급한다는 점에서 서브쿼리와 동일

차이점 : SELECT 문장들의 관계가 동등관계 이다.

SET연산자는 SQL 문장을 하나로 묶는 역할을 수행한다.

차집합, 교집합, 합집등으로 집합을 구분하는데, 이러한 개념을 SQL에서

구현하는데 사용되는 것이 SET연산자이다.

즉, SET연산자를 사용하여 테이블이나 뷰에 있는 데이터들을 대상으로

쿼리의 최종결과를 합집합 형태 혹은 차집합 형태 등으로 만들 수 있다.

ex) 하나의 쿼리 결과로 A라는 데이터 셋이 만들어졌다.

또다른 쿼리 결과로 B라는 데이터 셋이 만들어졌다.

A, B 데이터 셋에는 서로 공통된 데이터가 있을 수 있다.

이를 통해 합, 차, 교집합형태의 집합을 만들어 낼 수 있다.

**-- UNION**

(집합 A)

SELECT 1, 3, 4, 5, 7, 8

FROM DUAL;

(집합 B)

SELECT 2, 4, 5, 8, 9

FROM DUAL;

(두 개의 집합을 UNION으로 연결)

-- error) 질의의 결과 열의 수가 틀립니다.

SELECT 1, 3, 4, 5, 7, 8

FROM DUAL

UNION

SELECT 2, 4, 5, 8, 9

FROM DUAL;

-- SET연산자 사용시 SET연산자로 묶이는 SELECT 문장의 열의 개수가 반드시

같아야 한다.

SELECT 1, 3, 4, 5, 7, 8

FROM DUAL

UNION

SELECT 2, 4, 5, 8, 9, NULL

FROM DUAL;

==> 출력결과

1 3 4 5 7 8

----------------

1 3 4 5 7 8

2 4 5 8 9

-- SQL의 경우 단위가 개별 로우가 된다.

즉 집합론에서의 하나의 요소는 SQL에서의 하나의 로우에 해당된다.

위 결과는 집합 A의 로우와 집합B의 로우를 합쳐서 그 결과를 산출했기 때문에

합집합의 역할을 수행했다고 볼 수 있다.

--example) 독일에 살고 있는 사람들의 명단을 추출

단, CUSTOMERS 테이블에 고객의 명단이

EMPLOYEES 테이블에 사원 명단이 들어있다.

-- (1) EMPLOYEES 테이블에서 사원 명단을 추출한다.

SELECT emp.employee\_id, emp.first\_name, emp.last\_name

FROM employees emp,

departments dep,

locations loc

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id = loc.location\_id

AND loc.country\_id = 'DE';

-- (2) CUSTOMERS에서 독일에 사는 고객 명단을 추출한다.

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY';

-- (3) 두 문장을 UNION 연산자를 사용하여 합친다.

주의점 : SET 연산자를 사용할 때는 연결되는 두 문장의 컬럼 개수가 동일해야 한다.

개수만 같다면 컬럼명이 다르더라도 문제가 되지 않는다.

또한, 두 SELECT문장에서 반환되는 컬럼의 순서에 따른 데이터 타입도 동일한

유형이어야 한다.

SELECT emp.employee\_id, emp.first\_name, emp.last\_name

FROM employees emp,

departments dep,

locations loc

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id = loc.location\_id

AND loc.country\_id = 'DE'

UNION

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY';

-- 컬럼 순서를 변경하면 오류가 발생한다.

(또한, 두 SELECT문장에서 반환되는 컬럼의 순서에 따른 데이터 타입도 동일한

유형이어야 한다에 위배됨)

SELECT emp.first\_name, emp.employee\_id, emp.last\_name

FROM employees emp,

departments dep,

locations loc

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id = loc.location\_id

AND loc.country\_id = 'DE'

UNION

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY';

-- 컬럼의 동일 데이터 타입의 UNION 처리

정확히 같은 데이터 타입은 아니지만 같은 범주의 테이터 유형에 속하면 오류가 없다.

CREATE TABLE set1 (

id1 INTEGER,

names1 CHAR(5)

);

CREATE TABLE set2 (

id2 NUMBER,

names2 VARCHAR2(5)

);

INSERT INTO set1 VALUES ( 1, 'AAA');

INSERT INTO set2 VALUES ( 2, 'BB');

SELECT id1, names1

FROM set1

UNION

SELECT id2, names2

FROM set2;

-- SELECT 리스트에 명시하는 컬럼들이 동일한 데이터 타입과 개수만 일치시킨다면

두 개 이상의 문장들도 결합할 수 있다.

-- 위에서 구현된 SET2 테이블에 컬럼을 하나 더 추가한다.

ALTER TABLE set2

ADD last\_names VARCHAR(10);

UPDATE set2

SET last\_names = 'developer';

-- 3개의 문장을 연결한다.

SELECT emp.employee\_id , emp.first\_name, emp.last\_name

FROM employees emp,

departments dep,

locations loc

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id = loc.location\_id

AND loc.country\_id = 'DE'

UNION

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY'

UNION

SELECT id2, names2, last\_names

FROM set2;

-- 앞에서 UNION은 집합론의 합집합에 해당된다고 했다.

합집합은 A와 B 두개의 요소들을 모두 추출해 내는데, 공통적으로 갖고 있는 요소들은

하나만 포함되었다.

그런데 UNION이 처리하는 방식은 다르다.

UNION의 경우에는 공통값을 가진 로우를 한 번만 추출하기는 하는데, 이 공통값이 어느 한

데이터 셋에 포함되어 있을 경우에만 한번만 추출해 낸다.

만약, 공통값이 두 테이블에 모두 있을 경우에는 합집합과는 달리 두 번 추출하게 된다.

-- 실제 확인

DELETE set1;

DELETE set2;

INSERT INTO set1 ( id1, names1 ) VALUES (1, 'AAA');

INSERT INTO set1 ( id1, names1 ) VALUES (1, 'AAA');

INSERT INTO set1 ( id1, names1 ) VALUES (2, 'BBB');

INSERT INTO set2 ( id2, names2 ) VALUES (2, 'BBB');

INSERT INTO set2 ( id2, names2 ) VALUES (3, 'CCC');

INSERT INTO set2 ( id2, names2 ) VALUES (3, 'CCC');

SELECT id1, names1

FROM set1

UNION

SELECT id2, names2

FROM set2;

**--[ UNION ALL ]**

UNION과 거의 같은 개념이지만 한 가지 다른 점이 있다.

UNION ==> DISTINCT역할을 수행한다.

UNION ALL ==> DISTINCT 의 반대인 ALL의 역할을 수행한다.

-- UNION

SELECT id1, names1

FROM set1

UNION

SELECT id2, names2

FROM set2;

==> 출력결과

ID1 NAMES1

--------------------

1 AAA

2 BBB

2 BBB

3 CCC

--UNION ALL

SELECT id1, names1

FROM set1

UNION ALL

SELECT id2, names2

FROM set2;

==> 출력결과

ID1 NAMES1

--------------------

1 AAA

1 AAA

2 BBB

2 BBB

3 BBB

3 CCC

-- 모든 로우를 다 합쳐서 추출하게 된다.

**-- [ INTERSECT ]**

교차한다. 교집합을 의미

-- SELECT 문장들에서 컬럼의 데이터 타입과 순서, 개수가 일치해야 한다.

SELECT id1, names1

FROM set1

INTERSECT

SELECT id2, names2

FROM set2;

==> 출력결과

ID1 NAMES1

-----------------------------

0 rows selected

==> ???

-- 실행...

SELECT id1

FROM set1

INTERSECT

SELECT id2

FROM set2;

==> 출력결과

ID1 NAMES1

-----------------------------

2

1 rows selected

=> 정확히 원하는 결과가 산출되었다.

위의 예의 문제는 NAMES1과 NAMES2 컬럼 때문에 발생하였다.

SET1 테이블의 NAMES1 컬럼은 CHAR(5) 타입이며, SET2 테이블의 NAMES2 컬럼은 VARCHAR(5) 타입이다.

두 테이블에 저장된 공통값에 해당되는 'BBB' 라는 문자열은 INSERT 시에는 동일한 데이터를

입력했지만, 저장된 후에는 값이 다르게 된다.

-- 창고에 있는 상품과 주문내역에 있는 상품들 중 공통적으로 사용된 상품,

즉, 판매된 상품의 ID 값을 추출하는 쿼리임

SELECT product\_id

FROM inventories

INTERSECT

SELECT product\_id

FROM order\_items;

**--[ MINUS ]**

차집합과 동일하다.

A와 B 집합에 있어 A의 차집합은 A에만 속해 있는 원소를 말한다.

-- SET1의 차집합

SELECT id1

FROM set1

MINUS

SELECT id2

FROM set2;

==> 출력결과

ID1

-----------------------------

1

-- SET2의 차집합

SELECT id2

FROM set2

MINUS

SELECT id1

FROM set1;

==> 출력결과

ID2 NAMES1

-----------------------------

3

-- 값이 일치하지 않는 것은 공통값으로 보지 않는다.

SELECT id1, names1

FROM set1

MINUS

SELECT id2, names2

FROM set2;

==> 출력결과

ID1 NAMES1

-----------------------------

1 AAA

2 BBB

-- 창고에 보관되어 있으나 아직 주문되지 않은 상품을 조회해 보자.

즉, 창고 상품내역 정보가 들어 있는 INVENTORIES 테이블의 차집합을 구하는 것이다.

SELECT product\_id

FROM inventories

MINUS

SELECT product\_id

FROM order\_items;

-- 주문은 되었으나, 창고에 없는 상품내역을 조회하는 쿼리

SELECT product\_id

FROM order\_items

MINUS

SELECT product\_id

FROM inventories;

**-- [ SET 연산자와 NULL ]**

SET 연산자의 주의사항 : SET 연산자로 연결되는 SELECT 문장에서 컬럼들의 순서와 데이터 타입이

일치해야 한다.

만약, 컬럼의 일부로 NULL이 사용된다면?

NULL은 데이터 타입이 아니라 데이터 자체가 없는것이다.

따라서 SELECT문의 일부 컬럼이 NULL이 될 경우, 오라클은 더 이상

비교를 수행하지 않는다. 따라서 오류가 발생하지는 않고 NULL이 포함되어

결과가 추출된다.

SELECT 1 col1, 'FIRST' col2

FROM DUAL

UNION

SELECT 2 col1, NULL col2

FROM DUAL;

COL1 COL2

------------------------------------

1 FIRST

2

==> 문자형과 NULL이 비교된 경우의 결과임

-- 숫자형과 NULL

SELECT 1 col1, 'FIRST' col2

FROM DUAL

UNION

SELECT NULL col2, 'SECOND' col2

FROM DUAL;

COL1 COL2

------------------------------------

1 FIRST

SECOND

==> 숫자형도 NULL 이 포함되어 결과가 추출된다.

-- 날짜형과 NULL

SELECT 1 col1, SYSDATE col2

FROM DUAL

UNION

SELECT 2 col2, NULL col2

FROM DUAL;

COL1 COL2

------------------------------------

1 2007-08-31

2

==> 날짜형도 NULL이 포함되어 결과가 추출된다.

- 오라클 8i까지는 오류를 발생시켰는데 Oracle9i부터는 오류 없이

출력되게 변경되었다.

따라서 SET연산자 사용시 NULL은 특별히 신경쓰지 않아도 된다.

-- SET 연산자의 규칙과 제한사항들

0) SET연산자로 묶여지는 SELECT 문장의 컬럼 개수와 데이터 타입이

일치해야 한다.

1) SET연산자로 묶여지는 SELECT문장에서 BLOB, CLOB, BFILE, VARRAY,

그리고 중첩 테이블 타입인 컬럼은 사용할 수 없다.

-- CUSTOMERS 테이블에서 고객들의 전화번호 정보가 저장되어 있는

PHONE\_NUMBERS 컬럼이 있는데, 이 컬럼이 바로 중첩 테이블

타입의 컬럼이다.

중첩 테이블 타입의 컬럼은 컬럼 자체가 일반 테이블처럼

또 다른 컬럼들을 포함하고 있는 유형이다.

-- 에러 발생

SELECT phone\_numbers

FROM customers

WHERE nls\_territory = 'INDIA'

UNION

SELECT phone\_numbers

FROM customers

WHERE nls\_territory = 'ITALY';

2) UNION, INTERSECT, MINUS 연산자를 사용할 경우 SELECT 문장에서

LONG 타입 컬럼은 사용할 수 없다.

UNION ALL을 사용할 경우에는 사용할 수 있다.

3) SET 연산자와 함께 사용되는 SELECT 문장의 경우 FOR UPDATE 절을

사용할 수 없다.

4) SET 연산자를 사용할 경우 ORDER BY절은 문장 전체에서 맨 마지막

에 한 번만 사용할 수 있으며, SET 연산자로 연결되는 SELECT

문의 일부에서 서브쿼리가 사용된다면 서브쿼리 내부에서도

ORDER BY 절이 사용될 수 없다.

-- 에러 발생

ORDER BY절이 첫번째 SELECT 문장에서 사용됨

SELECT ID1, NAMES1

FROM SET1

ORDER BY 1

UNION

SELECT ID2, NAMES2

FROM SET2;

-- 에러 수정

SET연산자로 연결될 경우에는 맨 마지막에 한번만 ORDER BY절을

사용할 수 있다.

SELECT ID1, NAMES1

FROM SET1

UNION

SELECT ID2, NAMES2

FROM SET2

ORDER BY 1;

-- 서브쿼리가 사용될 경우, 서브쿼리내에서는 ORDER BY절을

사용할 수 없다.

-- 에러

SELECT emp.employee\_id, emp.first\_name, emp.last\_name

FROM employees emp,

departments dep

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id in ( SELECT loc.location\_id

FROM locations loc

WHERE loc.country\_id = 'DE'

ORDER BY location\_id)

UNION

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY';

-- 수정

ORDER BY절을 빼거나 맨 마지막에 사용해야 한다.

SELECT emp.employee\_id, emp.first\_name, emp.last\_name

FROM employees emp,

departments dep

WHERE emp.department\_id = dep.department\_id

AND dep.location\_id in ( SELECT loc.location\_id

FROM locations loc

WHERE loc.country\_id = 'DE' )

UNION

SELECT customer\_id, cust\_first\_name, cust\_last\_name

FROM customers

WHERE nls\_territory = 'GERMANY';

**3. 계층형 쿼리**

- 오라클에서만 지원하고 있는 아주 막강한 기능 중의 하나이다.

- 관계형 데이터 베이스에서 말하는 관계라는 의미는 계층형과는 상반된 개념이다.

따라서 일반적인 SQL을 사용해서 이러한 형태의 정보를 표현하기가 쉽지 않다.

- 계층형 : C++의 상속구조 / 파일의 디렉토리 구조

자료구조의 트리 구조

- 마스터 디테일 구조

계층형 구조는 아니다.

테이블간 부모 자식관계가 존재한다.

-- 계층형 쿼리

일반조인을 사용하여 계층형 쿼리를 구현할 수 있다.

CREATE TABLE BOM (

ITEM\_ID INTEGER NOT NULL,

PARENT\_ID INTEGER,

ITEM\_NAME VARCHAR2(20) NOT NULL,

ITEM\_QTY INTEGER,

PRIMARY KEY (ITEM\_ID));

INSERT INTO BOM VALUES ( 1001, NULL, '컴퓨터', 1);

INSERT INTO BOM VALUES ( 1002, 1001, '본체', 1);

INSERT INTO BOM VALUES ( 1003, 1001, '모니터', 1);

INSERT INTO BOM VALUES ( 1004, 1001, '프린터', 1);

INSERT INTO BOM VALUES ( 1005, 1002, 'Mother Board', 1);

INSERT INTO BOM VALUES ( 1006, 1002, '랜카드', 1);

INSERT INTO BOM VALUES ( 1007, 1002, 'Power Supply', 1);

INSERT INTO BOM VALUES ( 1008, 1005, 'RAM', 1);

INSERT INTO BOM VALUES ( 1009, 1005, 'CPU', 1);

INSERT INTO BOM VALUES ( 1010, 1005, '그래픽장치', 1);

INSERT INTO BOM VALUES ( 1011, 1005, '기타장치', 1);

==> 테이블의 개수가 하나이며 각각의 품목 정보들은 서로 부모와 자식 관계로

연결되어 있으므로 조인을 사용해야 한다.

자기 참조 성격이 있으므로 셀프조인을 사용한다.

문제점 : 루트노드에 해당하는 컴퓨터의 경우 가장 상위의 1레벨에 속하기 때

문에 PARENT\_ID 컬럼값이 NULL이된다.

따라서 최종 제품인 컴퓨터를 위해 외부조인도 필요하다.

SELECT bom1.item\_name,

bom1.item\_id,

bom2.item\_name parent\_item

FROM bom bom1, bom bom2

WHERE bom1.parent\_id = bom2.item\_id(+)

ORDER BY bom1.item\_id;

==> 실행결과?

계층적인 순서로 결과가 조회된다.(레벨순으로 결과가 나옴)

하지만 오해의 소지를 가지고 있슴

-- 오라클은 이러한 경우 계층적인 정보를 표현할 수 있게 특별한

문장을 지원한다. ==> START WITH.. CONNECT BY절임

SELECT LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id;

==> SELECT절의 ' '는 계층별로 품목을 나타내기 위해 품목명 앞에 공백을 삽입하는

용도로 사용하였다.

SELECT LPAD(' ', 4\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id

ORDER BY item\_id;

==> START WITH : 루트노드를 찾는 역할을 수행

LEVEL: 계층형 구조에서의 레벨(루트노드 => LEVEL1이된다.)

CONNECT BY : 부모와 자식노드들 간의 관계를 설정하는 부분

PRIOR : 계층형 쿼리에서만 사용되는 오라클 SQL연산자이다.

등호(=)와 동등한 레벨로 사용되는 연산자이며 CONNECT BY절에서

해당 컬럼의 부모 로우를 식별하는 데사용된다.

'본체' 항목의 경우 상위 노드가 컴퓨터이므로

PARENT\_ID, 항목에 컴퓨터의 ITEM\_ID 값을 갖고 있다.

따라서 이부분은 CONNECT\_BY 절에 명시하면 되고, 이�

PRIOR 키워드를 추가시킨다.

본체의 PARENT\_ID는 부모노드인 컴퓨터의 ITEM\_ID와 연결되므로

PRIOR키워드는 다음과 같이 PARENT\_ID가 아닌 ITEM\_ID앞에 붙여야 한다.

CONNECT BY PRIOR ITEM\_ID = PARENT\_ID( 0 )

CONNECT BY ITEM\_ID = PRIOR PARENT\_ID( X )

CONNECT BY PARENT\_ID = PRIOR ITEM\_ID ( 0 )

**-- [계층형 쿼리와 조인 ]**

계층형 쿼리에서의 조인사용

SELECT LEVEL,

LPAD(' ', 4\*(LEVEL -1)) || first\_name || ' ' || last\_name "성명"

FROM employees

START WITH MANAGER\_ID IS NULL

CONNECT BY manager\_id = PRIOR employee\_id;

==> 직위와 직급별로 차례대로 조회가 된다.

-- 직위별로 직위명까지 조회하는 쿼리문

SELECT b.job\_title "직위",

LPAD(' ', 4\*(LEVEL -1)) || a.first\_name || ' ' || a.last\_name "성명"

FROM employees a, jobs b

WHERE a.job\_id = b.job\_id

START WITH a.manager\_id IS NULL

CONNECT BY a.manager\_id = PRIOR a.employee\_id;

-- 각 사원들의 부서와 부서과 위치한 지역정보를 조회

SELECT b.job\_title "직위",

LPAD(' ', 4\*(LEVEL -1)) || a.first\_name || ' ' || a.last\_name "성명",

c.department\_name "부서",

d.city || ', ' || d.state\_province || ', ' || e.country\_name "부서위치"

FROM employees a,

jobs b,

departments c,

locations d,

countries e

WHERE a.job\_id = b.job\_id

AND a.department\_id = c.department\_id

AND c.location\_id = d.location\_id

AND d.country\_id = e.country\_id

START WITH a.manager\_id IS NULL

CONNECT BY a.manager\_id = PRIOR a.employee\_id;

-- 처리 순서 확인

계층형 쿼리에서 조인을 사용할 때 오라클 내부에서는 어떠한 순서로 처리될까?

- 1) 조인이 사용되었다면 가장 먼저 조인을 처리한다.

2) 그 다음으로 CONNECT BY 조건을 처리한다.

3) 마지막으로 나머지 조건(WHERE절에서 조인 이외의 조건)을 처리한다.

SELECT b.job\_title "직위",

LPAD(' ', 4\*(LEVEL -1)) || a.first\_name || ' ' || a.last\_name "성명",

c.department\_name "부서", a.salary

FROM employees a, jobs b, departments c

WHERE a.job\_id = b.job\_id

AND a.salary > 10000

AND a.department\_id = c.department\_id

START WITH a.manager\_id IS NULL

CONNECT BY a.manager\_id = PRIOR a.employee\_id;

-- 동일 확인 sample

SELECT LPAD(' ', 4\*(LEVEL -1)) || a.first\_name || ' ' || a.last\_name "성명",

c.department\_name "부서",

a.salary "급여"

FROM employees a, departments c

WHERE a.employee\_id = c.manager\_id

AND a.salary > 10000

START WITH a.manager\_id IS NULL

CONNECT BY a.manager\_id = PRIOR a.employee\_id;

**-- 계층형 쿼리의 확장**

-- 루트노드 찾기 : CONNECT\_BY\_ROOT

-- LEVEL 2이상인 품목들을 계층형 쿼리를 사용하여 조회

SELECT LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

WHERE LEVEL >= 2

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id;

-- 위의 쿼리는 최상위 품목인 컴퓨터만 제외하고 모든 하위품목이 조회되었다.

컴퓨털를 출력하려면 위의 쿼리는 불가능하다.

LEVEL1은 위의 쿼리로 가져올 수 없다.

아래의 쿼리를 수행한다면?

CONNECT\_BY\_ROOT를 제공한다. 단독으로 사용되지 못하고 일반 컬럼과

같이 사용된다.

조건식에서는 LEVEL >= 2이지만 루트노드가 출력된다.

계층형 쿼리는 내부적으로 항상 루트노드에 대한 정보를 가지고 있다.

SELECT LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names,

CONNECT\_BY\_ROOT item\_id root\_id,

CONNECT\_BY\_ROOT item\_name root\_name

FROM bom

WHERE LEVEL >= 2

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id;

-- 예상대로라면 ROOT\_ID와 ROOT\_NAME에 모두 컴퓨터에 해당하는 정보가

나와야 하는데, 본체에 대한 정보가 조회되었다.

(item\_id 1002 ==> 본체)

-- 루트노드를 식별하는 START\_WITH절에서 루트노드가 ITEM\_ID값이 102인 본체

문장을 지명했기 때문에 컴퓨터가 아닌 본체를 가져온것이다.

제일 먼저 처리되는 것이 WHERE절에 명시된 조인이며,

두번째가 계층형 연산, 그리고 마지막으로 처리되는 부분이 조인이 아닌

WHERE절이다.

SELECT LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names,

CONNECT\_BY\_ROOT item\_id root\_id,

CONNECT\_BY\_ROOT item\_name root\_name

FROM bom

START WITH item\_id = '1002'

CONNECT BY PRIOR item\_id = parent\_id;

-- 중복 참조값 찾기 (CONNECT\_BY\_ISCYCLE)

계층형 쿼리에서 사용되는 의사컬럼의 한종류

해당 로우의 항목이 자식노드를 갖고 있는데 동시에 그 자식노드가

다시 부모노드인지를 판별하는 위사커럼이다.

- 중복참조를 하는 경우 이를 판별하여 자식노드가 있을 경우 1을,

없을 경우 0을 반환한다.

SELECT item\_id, level,

LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH item\_id = 1005

CONNECT BY PRIOR item\_id = parent\_id;

UPDATE BOM

SET parent\_id = 1010

WHERE item\_id = 1005;

SELECT item\_id, level,

LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH item\_id = 1005

CONNECT BY PRIOR item\_id = parent\_id;

SELECT item\_id, level,

LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH item\_id = 1005

CONNECT BY NOCYCLE PRIOR item\_id = parent\_id;

SELECT item\_id, level,

LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names,

CONNECT\_BY\_ISCYCLE cycles

FROM bom

START WITH item\_id = 1005

CONNECT BY NOCYCLE PRIOR item\_id = parent\_id;

-- 리프노드 찾기(CONNECT\_BY\_ISLEAF)

SELECT item\_id, level,

CONNECT\_BY\_ISLEAF leafs,

LPAD(' ', 2\*(LEVEL-1)) || item\_name item\_names

FROM bom

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id;

-- 루트 찾아기기(SYS\_CONNECT\_BY\_PATH)

SELECT item\_id, level,

SYS\_CONNECT\_BY\_PATH (item\_id, '/') id\_path,

SYS\_CONNECT\_BY\_PATH (item\_name, '/') name\_path

FROM bom

START WITH parent\_id IS NULL

CONNECT BY PRIOR item\_id = parent\_id;

**-- 계층형 쿼리의 응용 실습**

-- 답변형 게시판

CREATE TABLE test\_boards (

id NUMBER(5) NOT NULL, -- ID

p\_id NUMBER(5) NULL, -- 상위 ID 값

subject VARCHAR2(30) NOT NULL, -- 게시판 제목

content VARCHAR2(100) NULL, -- 내용

create\_date DATE default sysdate); -- 생성일자

-- Sample 데이터 추가

INSERT INTO test\_boards VALUES (1, NULL, '오라클 책 추천좀..', '오라클을 공부하려는데, 책 좀 추천해 주세요', SYSDATE);

INSERT INTO test\_boards VALUES (2, NULL, '저도요..', '오라클 초보입니다..어떤 책을 보면 좋을까요?', SYSDATE);

INSERT INTO test\_boards VALUES (3, 1, '이 책이 어떨련지...', '한빛미디어에서 나온 뇌자극 시리즈가 좋습니다..', SYSDATE);

INSERT INTO test\_boards VALUES (4, 3, '동감..', '맞습니다..맞고요...정말 괜찮습니다...', SYSDATE);

INSERT INTO test\_boards VALUES (5, 1, '초보자라면 이 책도...', 'HEAD FIRST SQL .. 아직 한글판이 나오지 않은것 같은데..이 책 나오면 괜찮을 듯합니다. ', SYSDATE);

INSERT INTO test\_boards VALUES (6, 2, '이거 보세요', '그냥 메뉴얼 보세요..ㅋㅋ ', SYSDATE);

-- 계층형 쿼리문 작성 => 출력 결과 확인

SELECT rownum,

LPAD(' ', 2\*(LEVEL-1)) || subject

FROM test\_boards

START WITH p\_id IS NULL

CONNECT BY PRIOR id = p\_id;

-- 두번째 항목(ID 3)을 클릭했을 경우 발생되는 쿼리문

SELECT DECODE(CONNECT\_BY\_ISLEAF, 1, ' ' || CONTENT,

'[상위글] ' || CONTENT || CHR(13)) CONTESTS

FROM test\_boards

WHERE CONNECT\_BY\_ROOT ID = 1

START WITH P\_ID is null

CONNECT BY PRIOR id = p\_id

AND id = 3

ORDER BY LEVEL;

-- 위 코드) ID값을 3으로 주면 ID값이 3인 항목과 이 항목의 상위 노드를 추출한다.

SELECT id, p\_id, subject

FROM test\_boards

START WITH p\_id IS NULL

CONNECT BY PRIOR id = p\_id

AND id = 3;

-- 위 코드) WHERE절 추가

위 쿼리는 우리가 원하지 않는 조건이 포함되어 있다.

ID2...

단지 루트노드이기때문에 조회되었다. 따라서 WHERE조건에서

해당 항목을 제거한다.

즉, CONNECT\_BY\_ROOT 의사컬럼을 사용하여 루트노드의 ID값이 1인것만

조회한다.

SELECT id, p\_id, subject

FROM test\_boards

WHERE CONNECT\_BY\_ROOT id = 1

START WITH p\_id IS NULL

CONNECT BY PRIOR id = p\_id

AND id = 3;

-- 달력 만들기 SAMPLE 순차적으로 확인

SELECT TO\_DATE('200701', 'YYYYMM')

FROM DUAL;

SELECT make\_dates, LEVEL

FROM ( SELECT TO\_DATE('200701', 'YYYYMM') make\_dates

FROM DUAL )

CONNECT BY LEVEL <= 31;

SELECT (make\_dates + LEVEL - 1) dates

FROM ( SELECT TO\_DATE('200701', 'YYYYMM') make\_dates

FROM DUAL )

CONNECT BY LEVEL <= 31;

SELECT DECODE( TO\_CHAR(dates,'D'), 1, TO\_CHAR(dates, 'DD')) 일,

DECODE( TO\_CHAR(dates,'D'), 2, TO\_CHAR(dates, 'DD')) 월,

DECODE( TO\_CHAR(dates,'D'), 3, TO\_CHAR(dates, 'DD')) 화,

DECODE( TO\_CHAR(dates,'D'), 4, TO\_CHAR(dates, 'DD')) 수,

DECODE( TO\_CHAR(dates,'D'), 5, TO\_CHAR(dates, 'DD')) 목,

DECODE( TO\_CHAR(dates,'D'), 6, TO\_CHAR(dates, 'DD')) 금,

DECODE( TO\_CHAR(dates,'D'), 7, TO\_CHAR(dates, 'DD')) 토

FROM (

SELECT (make\_dates + LEVEL - 1) dates

FROM ( SELECT TO\_DATE('200701', 'YYYYMM') make\_dates

FROM DUAL )

CONNECT BY LEVEL <= 31

);

SELECT TO\_CHAR(DATES, 'W'),

MIN(DECODE( TO\_CHAR(dates,'D'), 1, TO\_CHAR(dates, 'DD'))) 일,

MIN(DECODE( TO\_CHAR(dates,'D'), 2, TO\_CHAR(dates, 'DD'))) 월,

MIN(DECODE( TO\_CHAR(dates,'D'), 3, TO\_CHAR(dates, 'DD'))) 화,

MIN(DECODE( TO\_CHAR(dates,'D'), 4, TO\_CHAR(dates, 'DD'))) 수,

MIN(DECODE( TO\_CHAR(dates,'D'), 5, TO\_CHAR(dates, 'DD'))) 목,

MIN(DECODE( TO\_CHAR(dates,'D'), 6, TO\_CHAR(dates, 'DD'))) 금,

MIN(DECODE( TO\_CHAR(dates,'D'), 7, TO\_CHAR(dates, 'DD'))) 토

FROM (

SELECT (make\_dates + LEVEL - 1) dates

FROM ( SELECT TO\_DATE('200701', 'YYYYMM') make\_dates

FROM DUAL )

CONNECT BY LEVEL <= 31

)

GROUP BY TO\_CHAR(DATES, 'W');

SELECT MIN(DECODE( TO\_CHAR(dates,'D'), 1, TO\_CHAR(dates, 'DD'))) 일,

MIN(DECODE( TO\_CHAR(dates,'D'), 2, TO\_CHAR(dates, 'DD'))) 월,

MIN(DECODE( TO\_CHAR(dates,'D'), 3, TO\_CHAR(dates, 'DD'))) 화,

MIN(DECODE( TO\_CHAR(dates,'D'), 4, TO\_CHAR(dates, 'DD'))) 수,

MIN(DECODE( TO\_CHAR(dates,'D'), 5, TO\_CHAR(dates, 'DD'))) 목,

MIN(DECODE( TO\_CHAR(dates,'D'), 6, TO\_CHAR(dates, 'DD'))) 금,

MIN(DECODE( TO\_CHAR(dates,'D'), 7, TO\_CHAR(dates, 'DD'))) 토

FROM (

SELECT (make\_dates + LEVEL - 1) dates

FROM ( SELECT TO\_DATE('200701', 'YYYYMM') make\_dates

FROM DUAL )

CONNECT BY (make\_dates + LEVEL - 1) <= LAST\_DAY(make\_dates)

)

GROUP BY DECODE(TO\_CHAR(DATES, 'D'), 1, TO\_CHAR(DATES, 'W') + 1, TO\_CHAR(DATES, 'W'))

ORDER BY DECODE(TO\_CHAR(DATES, 'D'), 1, TO\_CHAR(DATES, 'W') + 1, TO\_CHAR(DATES, 'W'));