<ANSI Join>

Inner Join

Self Join

Inner Join과 같은 경우, Data가 매칭되는 것이 없으면 누락되는 경우가 있다.

이런 경우에는 누락되는 것을 방지하기 위해, Data가 부족한 쪽에 대해 (+) 연산자를 추가하여 Outer Join을 사용한다.

ANSI Join : 어느 DB에서나 똑같이 쓰이는 표준 Join

카다시안 Join -> Cross Join으로 만듦

Table과 Table 사이에 Cross만 넣어주면 됨

Natural Join (Equi Join) –

Using Join (Equi Join) – Natural Join보다 Column을 제약시켜 보다 제한적으로 Join을 사용하고 싶을 때

Natural이나 Using Join은 Column 명이 같아야만 처리가 가능.

Join On – 가장 유연하게 Join 사용 가능

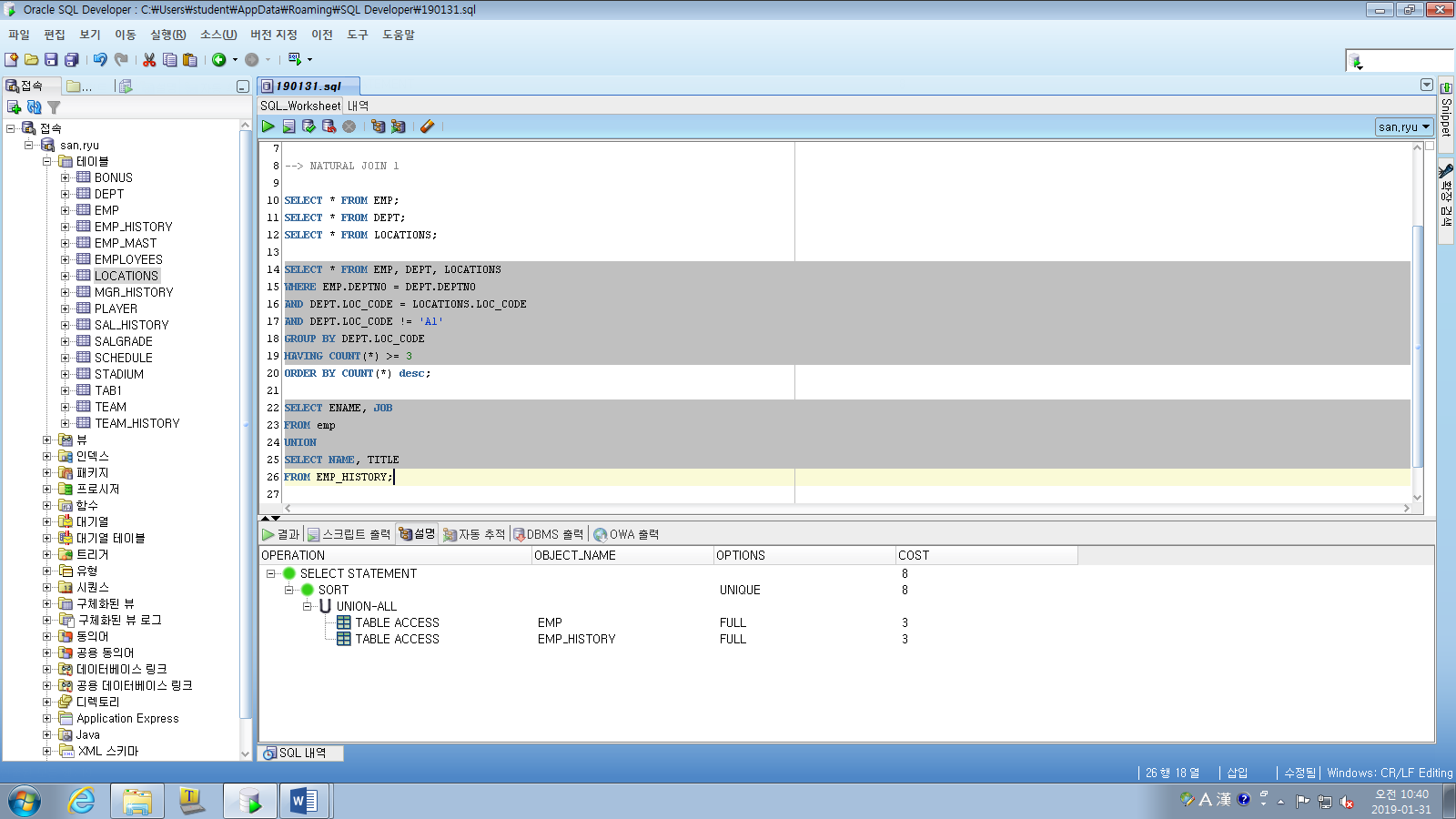
<SET 연산자>

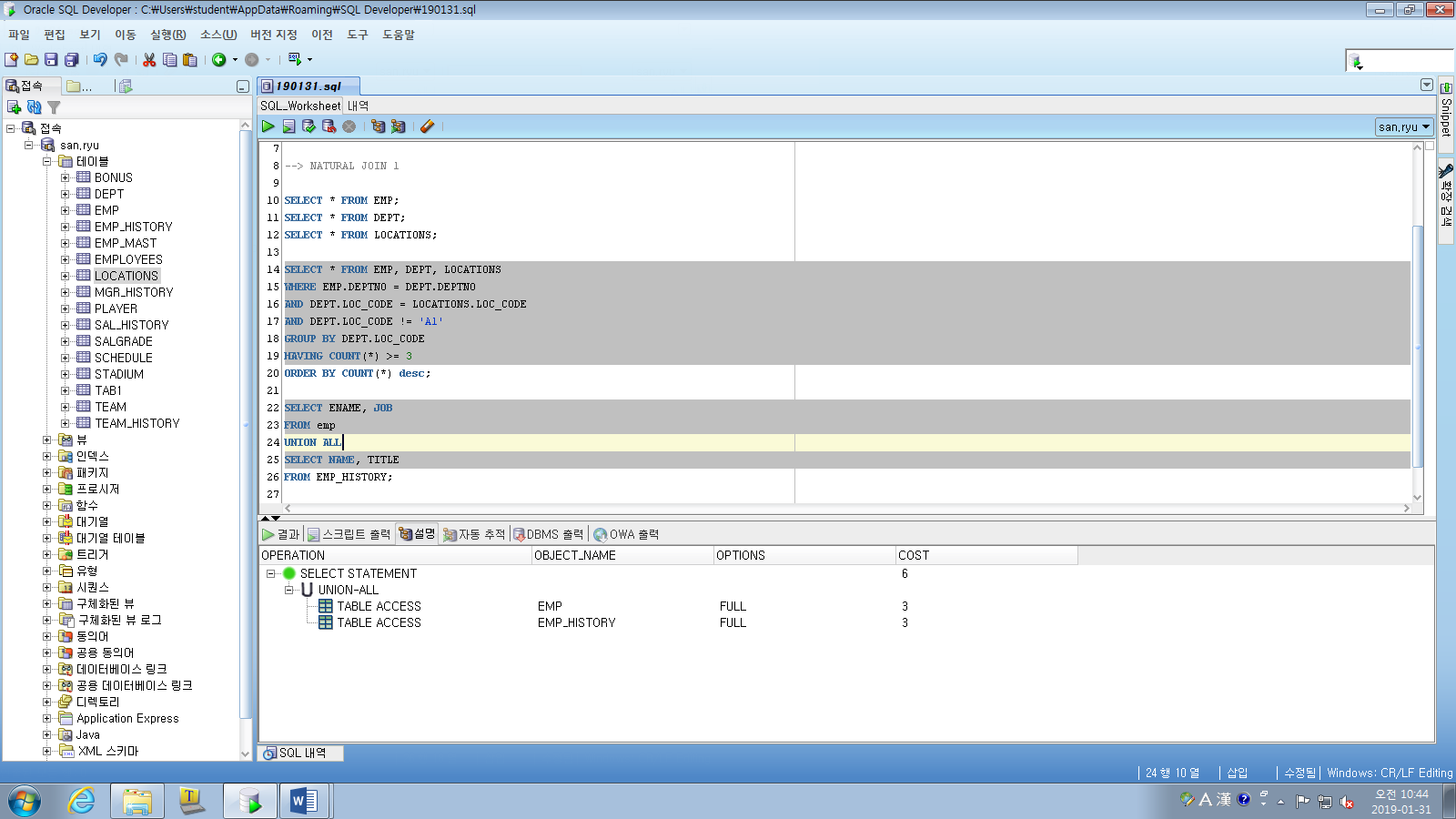
<UNION과 UNION ALL>

중복의 유무.

ALL은 중복을 배제하지 않음

두 테이블간 공통 집합이 없는 경우 UNION ALL을 사용하는 것이 유리하다. 왜냐면, SORTING과정을 거치지 않기 때문





INTERSECT

ㅁ SQL Command Line 실행

Connect /as sysdba

Create user sql01 identified by sql01;

Grant connect, resource, dba to sql01;

Connect sql01/sql01

@c:\create1.sql

<서브쿼리>

문제 해결을 주 목적으로 사용됨

EX) John의 급여를 알고 싶은 경우,

SELECT SAL FROM EMP

WHERE ENAME = ‘JONES’;

>>> SAL : 2975

그러면 John보다 급여를 많이 받는 사람은?

SELECT SAL FROM EMP

WHERE SAL >= 2975

>> 두번의 Query가 필요

그러므로,

SELECT EMPNO, ENAME FROM EMP

WHERE SAL > (SELECT SAL

FROM EMP

WHERE ENAME = ‘JONES’);

서브쿼리, 또는 이너쿼리라고 불림

바깥쪽은 메인쿼리 또는 아우터쿼리라고 함

SUBQUERY의 종류

* Return 종류에 따라
  + 단일 컬럼(Single Column) 서브쿼리
  + 다중 컬럼(Multi Column)
  + 단일행(Single Row)
  + 다중행(Multi Row)
* 동작 방식에 따른 분류
  + NESTED 서브쿼리 : 혼자 독립적으로 단 한번만 실행됨.
  + 상호연관(Correlated) 서브쿼리 : 혼자 독립적으로 수행되지 않으며, Main Query Table의   
     특정 Column을 참조하거나, 해당 Column과 관련된   
     조건을 가지게 됨.
* 기술 위치에 따른 분류
  + Scalar 서브쿼리 (SELECT절) : 내부적으로 Cache 기능을 사용. 동일한 Subquery에 대해서는 반복수행을 하지 않음. 사용에 따라 함수보다 더 효율적인 성능을 보임.  
    \* 단일 값만 리턴이 되며, 2개 이상 값이 리턴되는 경우 오류가 난다.  
    Outer Join을 표현하기에 적합한 Subquery
  + Inline-View (FROM절) : 가상 테이블
  + 서브쿼리 (WHERE절)

=(EQUAL)은 IN으로 대체시켜도 됨.

Equal은 오류가 날 확률이 있으나, IN은 나중에 오류가 날 일이 없음.

ANY / ALL은 앞에 비교 연산자와 함께 쓰이기 때문에,

|  |  |
| --- | --- |
| ANY – OR | EQUAL + OR + EQUAL +OR = IN |
| EX) sal > ANY(1000, 2000, 3000)  >>  sal > 1000  or sal > 2000  or sal > 3000  >>  sal > (SELECT MIN(sal) ~) | EX) sal > ANY(1000, 2000, 3000)  >>  sal > 1000  or sal > 2000  or sal > 3000  >>  sal > (SELECT MAX(sal) ~) |
| ALL – AND |  |
| EX) sal > ANY(1000, 2000, 3000)  >>  sal > 1000  AND sal > 2000  AND sal > 3000  >>  sal > (SELECT MAX(sal) ~) | EX) sal > ANY(1000, 2000, 3000)  >>  sal > 1000  AND sal > 2000  AND sal > 3000  >>  sal > (SELECT MIN(sal) ~) |

P.160 예제

Nested Subquery

SELECT 사원명, 사원 번호, 연봉

FROM 사원

WHERE 연봉 >= (SELECT AVG(연봉)

FROM 사원)ㅣ

서브쿼리를 읽어서 메인쿼리로 참조하는 작업을 단 1번 수행

Correlated Subquery

SELECT M.사원명, M.사원번호, M.연봉

FROM 사원 M

WHERE M.연봉 > (SELECTED AVG(S.연봉)

FROM 사원 S

WHERE S,부서번호 = M,부서번호);

Main Query를 수행할 때마다 매번 수행

P.165 예제

전체 EMP 수가 14번이므로, 해당 Subquery는 14번 수행된다.

속도가 크게 떨어지진 않으나, EMP Table에 대한 평균(AVG Function)을 14번 구하게 된다.

1. INLINE VIEW가 중첩되어, 안쪽에서부터 절차적으로 해석되는 Subquery 형태가 많은데,

이는 원하는 로직을 구현하기 위한 목적.

1. FROM절에서 Table간 Join을 시키면 되지만, Subquery를 통해 축약된 Table을 Join시키는 방식으로 Data 처리량을 줄이는 목적

상호연관(Correlated) 서브쿼리 사용시 주의사항

1. 상호연관 사용시  
   SELECT M.사원명, M.사원번호, M.연봉  
   FROM 사원 M  
   WHERE M.연봉 > (SELECT AVG(S.연봉)  
    FROM 사원 S  
    WHERE S.부서번호 = M.부서번호)  
   -- 수행 8만번. (수행시 동일 데이터에 대한 AVG를 구하기 위해 8만번 중복 접근)
2. Inline View 사용시  
   SELECT M.사원명, M.사원번호, M.연봉  
   FROM 사원 M, (SELECT 부서번호, AVG(연봉) 부서평균연봉  
    FROM 사원  
    GROUP BY 부서번호) S  
   WHERE M.연봉 > S.부서평균연봉  
   AND M.부서번호 = S.부서번호;  
   -- 8만건을 1번 수행 (FULL SCAN)

INLINE VIEW : TOP-N QUERY (상위 N명)

Scalar Subqury

1. Main Query를 수행한 후 Scalar Subquery에 필요한 값을 제공.
2. Scalar SubQuery를 수행하기 위해 필요한 데이터가 들어있는 블록을 메모리로 로딩
3. Main Query의 첫번째 행으로부터 주어진 조건을 가지고 필요한 값을 검색 후 이 결과를 입력값과 출력값으로 메모리 내의 Query Execution Cache에 저장.  
   여기서 입력값은 Main Query로부터 주어진 값이고, 출력값은 Scalar Sub Query를 수행하여 나온 결과값.  
   이 값을 저장하는 캐쉬를 지정하는 파라미터 : query\_execution\_cache\_max\_size
4. 다음행 조건이 Main Query에서 Scalar Subquery로 들어오면 Hash 함수를 이용해서 해당 값이 캐쉬에 존재하는지 찾고, 있으면 즉시 결과값을 출력하고 없으면 검색을 해서 해당 값을 찾은 후에 다시 메모리에 캐싱
5. Main Query의 모든 행에 대해서 끝날 때까지 반복 (때문에, Main Query에 해당하는 Table의 크기가 큰 경우, 성능상 문제가 생길 수 있다.)

스칼라 서브쿼리는 주로 코드성 테이블을 조회할 경우에 적합함.

EXIST 연산의 특징, 테이블을 읽다가 조건에 맞는 데이터가 있으면 해당 이후로 더 이상 데이터를 읽지 않으며, 다음 조건으로 넘어감.

EXIST의 Subquery의 SELECT문에 1이나 X와 같은 임의의 리터럴 값을 부여한다. (구색을 맞추는 문장) 존재 여부만 확인하도록 해야한다. (Column도 정의할 필요가 없다.) Count 등을 넣게되면, 해당 계산을 위해 Table을 끝까지 읽기 때문에 성능 저하의 요인이 될 수 있다.

<WITH>

일단 선언만 해두면, WITH절의 원래 구문을 실수로 다시 쓰더라도 WITH절 처리를 해주기 때문에, 더욱 효율적으로 사용 가능.

<분석함수>

ROWS UNBOUNDED PRECEDING / CURRENT ROW / N PRECEDING

ROWS BETWEEN UNBOUNDED PRECEDING / CURRENT ROW / N PRECEDING / N FOLLOWING

PRECEDING : 현재행 기준으로 앞행

N PRECEDING : 현재 행 기준으로 N개 앞행

3 PRECEDING 등…

FOLLWING : 현재 행 기준으로 뒷행

N FOLLOWING : 현재 행 기준으로 N개 뒷행

3 FOLLOWING 등…

ROWS – 행의 개수를 기준으로 분석이 가능하다.

그러나. RANGE(값의 범위)에 따라 분석을 하는 경우도 가능하다.