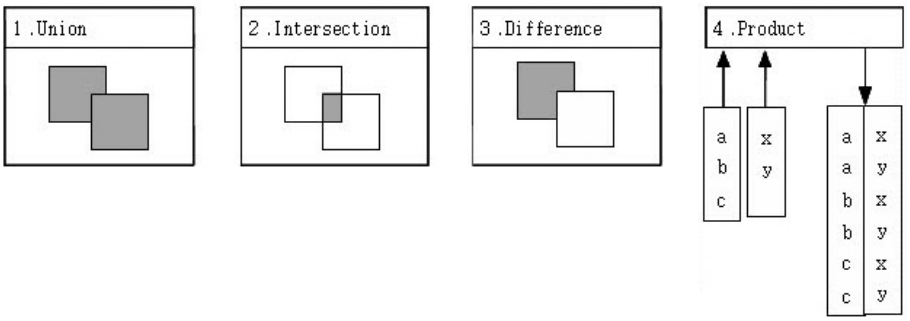
1. STANDARD SQL 개요

1970년: Dr. E.F.Codd 관계형 DBMS(Relational DB) 논문 발표 1974년: IBM SQL 개발 1979년: Oracle 상용 DBMS 발표 1980년: Sybase SQL Server 발표 (이후 Sybase ASE로 개명) 1983년: IBM DB2 발표 1986년: ANSI/ISO SQL 표준 최초 제정 (SQL-86, SQL1) 1992년: ANSI/ISO SQL 표준 개정 (SQL-92, SQL2) 1993년: MS SQL Server 발표 (Windows OS, Sybase Code 활용) 1999년: ANSI/ISO SQL 표준 개정 (SQL-99, SQL3) 2003년: ANSI/ISO SQL 표준 개정 (SQL-2003) 2008년: ANSI/ISO SQL 표준 개정 (SQL-2008)

국내뿐만 아니라 전 세계적으로 많이 사용되고 있는 관계형 데이터베이스의 경우 오브젝트 개념을 포함한 여러 새로운 기능들이 꾸준히 개발되고 있으며, 현재 기업형 DBMS는 순수 관계형 데이터베이스가 아닌 객체 지원 기능이 포함된 객체관계형(Object Relational) 데이터베이스를 대부분 사용하고 있다. 현재 우리가 사용하는 많은 시스템의 두뇌 역할을 하는 관계형 데이터베이스를 유일하게 접속할 수 있는 언어가 바로 SQL이다. 사용자와 개발자 입장에서는 SQL의 진화 및 변화가 가장 큰 관심 내용인데, 초창기 SQL의 기본 기능을 정리했던 최초의 SQL-86 표준과 관계형 DBMS의 폭발적인 전성기를 주도했던 ANSI/ISO SQL2 세대를 지나면서 많은 기술적인 발전이 있었다. 그러나, ANSI/ISO SQL2의 경우 표준 SQL에 대한 명세가 부족한 부분이 있었고, DBMS 벤더 별로 문법이나 사용되는 용어의 차이가 너무 커져서 상호 호환성이나 SQL 학습 효율이 많이 부족한 문제가 발생하였다. 이에 향후 SQL에서 필요한 기능을 정리하고 호환 가능한 여러 기준을 제정한 것이 1999년에 정해진 ANSI/ISO SQL3이다. 이후 가장 먼저 ANSI/ISO SQL3의 기능을 시현한 것이 Oracle의 8i/9i 버전이라고 할 수 있다. 참고로 2003년에 ANSI/ISO SQL 기준이 소폭 추가 개정되었고 현재 사용되는 데이터베이스는 대부분 SQL-2003 표준을 기준으로 하고 있다. 다른 벤더의 DBMS도 2006년 이후 발표된 버전에서 ANSI/ISO SQL-99와 SQL-2003의 핵심적인 기능은 만족스러운 수준으로 구현된 것으로 평가 받고 있다. 마지막으로 2008년에 진행된 추가 개정 내용은 아직 사용자 레벨에 큰 영향을 미치지 않고 있다. 아직도 벤더별로 일부 기능의 개발이 진행 중인 경우도 있고 벤더별 특이한 기술 용어는 여전히 호환이 안 되고 있지만, ANSI/ISO SQL 표준을 통해 STANDARD JOIN을 포함한 많은 기능이 상호 벤치마킹하고 발전하면서 DBMS 간에 평준화를 이루어 가고 있다고 볼 수 있다. 예를 들면, IBM DB2나 SYBASE ASE DBMS는 과거 버전부터 CASE 기능이나 FULL OUTER JOIN 기능을 지원하였지만, Oracle DBMS는 양쪽(FULL) OUTER JOIN의 경우 (+) 표시를 이용한 두 개의 SQL 문장을 UNION 오퍼레이션으로 처리하거나, CASE 기능을 구현하기 위해 DECODE 함수를 복잡하게 구현해야 하는 불편함이 있었다. 이런 불편 사항은 Oracle에서 표준 SQL에 포함된 CASE 기능과 FULL OUTER JOIN 기능을 추가함으로써 문제가 해결되었다.(참고로, Oracle DECODE 함수가 CASE 기능보다 장점도 있으므로 Oracle 사용자는 요구사항에 따라 DECODE나 CASE 함수를 선택할 수 있다.) 결과적으로 사용자 입장에서는 ANSI/ISO SQL의 새로운 기능들을 사용함으로써 보다 쉽게 데이터를 추출하거나 SQL 튜닝의 효과를 함께 얻을 수 있게 되었다. 대표적인 ANSI/ISO 표준 SQL의 기능은 다음 내용을 포함한다.

- STANDARD JOIN 기능 추가 (CROSS, OUTER JOIN 등 새로운 FROM 절 JOIN 기능들) - SCALAR SUBQUERY, TOP-N QUERY 등의 새로운 SUBQUERY 기능들 - ROLLUP, CUBE, GROUPING SETS 등의 새로운 리포팅 기능 - WINDOW FUNCTION 같은 새로운 개념의 분석 기능들

가. 일반 집합 연산자



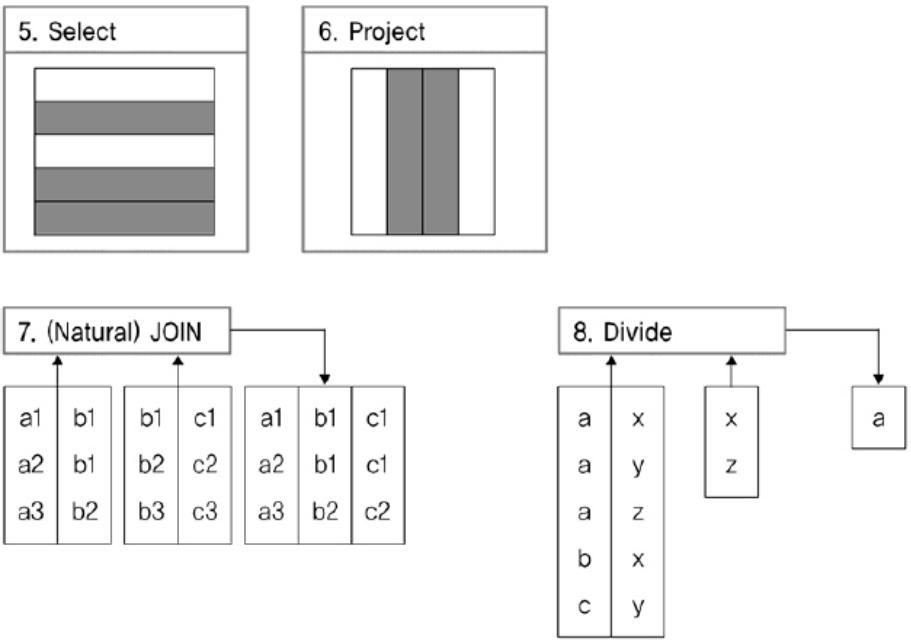
[그림 Ⅱ-2-1] E.F.CODD 일반 집합 연산자

현재 사용하는 SQL의 많은 기능이 관계형 데이터베이스의 이론을 수립한 E.F.Codd 박사의 논문에 언급이 되어 있다. 논문에 언급된 8가지 관계형 대수는 다시 각각 4개의 일반 집합 연산자와 순수 관계 연산자로 나눌 수 있으며, 관계형 데이터베이스 엔진 및 SQL의 기반 이론이 되었다. 일반 집합 연산자를 현재의 SQL과 비교하면,

1. UNION 연산은 UNION 기능으로, 2. INTERSECTION 연산은 INTERSECT 기능으로, 3. DIFFERENCE 연산은 EXCEPT(Oracle은 MINUS) 기능으로, 4. PRODUCT 연산은 CROSS JOIN 기능으로 구현되었다.

첫 번째, UNION 연산은 수학적 합집합을 제공하기 위해, 공통 교집합의 중복을 없애기 위한 사전 작업으로 시스템에 부하를 주는 정렬 작업이 발생한다. 이후 UNION ALL 기능이 추가되었는데, 특별한 요구 사항이 없다면 공통집합을 중복해서 그대로 보여 주기 때문에 정렬 작업이 일어나지 않는 장점을 가진다. 만일 UNION과 UNION ALL의 출력 결과가 같다면, 응답 속도 향상이나 자원 효율화 측면에서 데이터 정렬 작업이 발생하지 않는 UNION ALL을 사용하는 것을 권고한다. 두 번째, INTERSECTION은 수학의 교집합으로써 두 집합의 공통집합을 추출한다. 세 번째, DIFFERENCE는 수학의 차집합으로써 첫 번째 집합에서 두 번째 집합과의 공통집합을 제외한 부분이다. 대다수 벤더는 EXCEPT를, Oracle은 MINUS 용어를 사용한다. (SQL 표준에는 EXCEPT로 표시되어 있으며, 벤더에서 SQL 표준 기능을 구현할 때 다른 용어를 사용하는 것은 현실적으로 허용되고 있다.) 네 번째, PRODUCT의 경우는 CROSS(ANIS/ISO 표준) PRODUCT라고 불리는 곱집합으로, JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말한다. 양쪽 집합의 M*N 건의 데이터 조합이 발생하며, CARTESIAN(수학자 이름) PRODUCT라고도 표현한다.

나. 순수 관계 연산자



[그림 Ⅱ-2-2] E.F.CODD 순수 관계 연산자

순수 관계 연산자는 관계형 데이터베이스를 구현하기 위해 새롭게 만들어진 연산자이다. 순수 관계 연산자를 현재의 SQL 문장과 비교하면 다음과 같다.

5. SELECT 연산은 WHERE 절로 구현되었다. 6. PROJECT 연산은 SELECT 절로 구현되었다. 7. (NATURAL) JOIN 연산은 다양한 JOIN 기능으로 구현되었다. 8. DIVIDE 연산은 현재 사용되지 않는다.

다섯 번째, SELECT 연산은 SQL 문장에서는 WHERE 절의 조건절 기능으로 구현이 되었다. (SELECT 연산과 SELECT 절의 의미가 다름을 유의하자.) 여섯 번째, PROJECT 연산은 SQL 문장에서는 SELECT 절의 칼럼 선택 기능으로 구현이 되었다. 일곱 번째, JOIN 연산은 WHERE 절의 INNER JOIN 조건과 함께 FROM 절의 NATURAL JOIN, INNER JOIN, OUTER JOIN, USING 조건절, ON 조건절 등으로 가장 다양하게 발전하였다. 여덟 번째, DIVIDE 연산은 나눗셈과 비슷한 개념으로 왼쪽의 집합을 'XZ'로 나누었을 때, 즉 'XZ'를 모두 가지고 있는 'A'가 답이 되는 기능으로 현재 사용되지 않는다. 관계형 데이터베이스의 경우 요구사항 분석, 개념적 데이터 모델링, 논리적 데이터 모델링, 물리적 데이터 모델링 단계를 거치게 되는데, 이 단계에서 엔터티 확정 및 정규화 과정, 그리고 M:M (다대다) 관계를 분해하는 절차를 거치게 된다. 특히 정규화 과정의 경우 데이터 정합성과 데이터 저장 공간의 절약을 위해 엔터티를 최대한 분리하는 작업으로, 일반적으로 3차 정규형이나 보이스코드 정규형까지 진행하게 된다. 이런 정규화를 거치면 하나의 주제에 관련 있는 엔터티가 여러 개로 나누어지게 되고, 이 엔터티들이 주로 테이블이 되는데 이렇게 흩어진 데이터를 연결해서 원하는 데이터를 가져오는 작업이 바로 JOIN이라고 할 수 있다. 관계형 데이터베이스에 있어서 JOIN은 SQL의 가장 중요한 기능이므로 충분히 이해할 필요가 있다.

2. FROM 절 JOIN 형태

ANSI/ISO SQL에서 표시하는 FROM 절의 JOIN 형태는 다음과 같다.

- INNER JOIN - NATURAL JOIN - USING 조건절 - ON 조건절 - CROSS JOIN - OUTER JOIN

ANSI/ISO SQL에서 규정한 JOIN 문법은 WHERE 절을 사용하던 기존 JOIN 방식과 차이가 있다. 사용자는 기존 WHERE 절의 검색 조건과 테이블 간의 JOIN 조건을 구분 없이 사용하던 방식을 그대로 사용할 수 있으면서, 추가된 선택 기능으로 테이블 간의 JOIN 조건을 FROM 절에서 명시적으로 정의할 수 있게 되었다. INNER JOIN은 WHERE 절에서부터 사용하던 JOIN의 DEFAULT 옵션으로 JOIN 조건에서 동일한 값이 있는 행만 반환한다. DEFAULT 옵션이므로 생략이 가능하지만, CROSS JOIN, OUTER JOIN과는 같이 사용할 수 없다. NATURAL JOIN은 INNER JOIN의 하위 개념으로 NATURAL JOIN은 두 테이블 간의 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI(=) JOIN을 수행한다. NATURAL INNER JOIN이라고도 표시할 수 있으며, 결과는 NATURAL JOIN과 같다. 새로운 SQL JOIN 문장 중에서 가장 중요하게 기억해야 하는 문장은 ON 조건절을 사용하는 경우이다. 과거 WHERE 절에서 JOIN 조건과 데이터 검증 조건이 같이 사용되어 용도가 불분명한 경우가 발생할 수 있었는데, WHERE 절의 JOIN 조건을 FROM 절의 ON 조건절로 분리하여 표시함으로써 사용자가 이해하기 쉽도록 한다. ON 조건절의 경우 NATURAL JOIN처럼 JOIN 조건이 숨어 있지 않고, 명시적으로 JOIN 조건을 구분할 수 있고, NATURAL JOIN이나 USING 조건절처럼 칼럼명이 똑같아야 된다는 제약 없이 칼럼명이 상호 다르더라도 JOIN 조건으로 사용할 수 있으므로 앞으로 가장 많이 사용될 것으로 예상된다. 다만, FROM 절에 테이블이 많이 사용될 경우 다소 복잡하게 보여 가독성이 떨어지는 단점이 있다. 그런 측면에서 SQL Server의 경우 ON 조건절만 지원하고 NATURAL JOIN과 USING 조건절을 지원하지 않고 있는 것으로 보인다. 본 가이드는 ANSI/ISO SQL 기준에 NATURAL JOIN과 USING 조건절이 표시되어 있으므로 이 부분도 설명을 하도록 한다.

3. INNER JOIN

INNER JOIN은 OUTER(외부) JOIN과 대비하여 내부 JOIN이라고 하며 JOIN 조건에서 동일한 값이 있는 행만 반환한다. INNER JOIN 표시는 그 동안 WHERE 절에서 사용하던 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다.

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제] WHERE 절 JOIN 조건 SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO; 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다. FROM 절 JOIN 조건 SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME FROM EMP INNER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO; INNER는 JOIN의 디폴트 옵션으로 아래 SQL문과 같이 생략 가능하다. SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME FROM EMP JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

[실행 결과] DEPTNO EMPNO ENAME DNAME ------ ----- ------ --------- 20 7369 SMITH RESEARCH 30 7499 ALLEN SALES 30 7521 WARD SALES 20 7566 JONES R ESEARCH 30 7654 MARTIN SALES 30 7698 BLAKE SALES 10 7782 CLARK ACCOUNTING 20 7788 SCOTT RESEARCH 10 7839 KING ACCOUNTING 30 7844 T URNER SALES 20 7876 ADAMS RESEARCH 30 7900 JAMES SALES 20 7902 FORD RESEARCH 10 7934 MILLER ACCOUNTING 14개의 행이 선택되었다.

위에서 사용한 ON 조건절에 대해서는 뒤에서 추가 설명하도록 한다.

4. NATURAL JOIN

NATURAL JOIN은 두 테이블 간의 동일한 이름을 갖는 모든 칼럼들에 대해 EQUI(=) JOIN을 수행한다. NATURAL JOIN이 명시되면, 추가로 USING 조건절, ON 조건절, WHERE 절에서 JOIN 조건을 정의할 수 없다. 그리고, SQL Server에서는 지원하지 않는 기능이다.

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제] SELECT DEPTNO, EMPNO, ENAME, DNAME FROM EMP NATURAL JOIN DEPT;

[실행 결과] DEPTNO EMPNO ENAME DNAME ------ ------ ------ ------ 20 7369 SMITH RESEARCH 30 7499 ALLEN SALES 30 7521 WARD SALES 20 7566 JONES RE SEARCH 30 7654 MARTIN SALES 30 7698 BLAKE SALES 10 7782 CLARK ACCOUNTING 20 7788 SCOTT RESEARCH 10 7839 KING ACCOUNTING 30 7844 TU RNER SALES 20 7876 ADAMS RESEARCH 30 7900 JAMES SALES 20 7902 FORD RESEARCH 10 7934 MILLER ACCOUNTING 14개의 행이 선택되었다.

위 SQL은 별도의 JOIN 칼럼을 지정하지 않았지만, 두 개의 테이블에서 DEPTNO라는 공통된 칼럼을 자동으로 인식하여 JOIN을 처리한 것이다. JOIN에 사용된 칼럼들은 같은 데이터 유형이어야 하며, ALIAS나 테이블 명과 같은 접두사를 붙일 수 없다.

[예제] SELECT EMP.DEPTNO, EMPNO, ENAME, DNAME FROM EMP NATURAL JOIN DEPT; ERROR: NATURAL JOIN에 사용된 열은 식별자를 가질 수 없음

NATURAL JOIN은 JOIN이 되는 테이블의 데이터 성격(도메인)과 칼럼명 등이 동일해야 하는 제약 조건이 있다. 간혹 모델링 상의 부주의로 인해 동일한 칼럼명이더라도 다른 용도의 데이터를 저장하는 경우도 있으므로 주의해서 사용해야 한다.

[예제] 아래 '*' 와일드카드처럼 별도의 칼럼 순서를 지정하지 않으면 NATURAL JOIN의 기준이 되는 칼럼 들이 다른 칼럼보다 먼저 출력된다. (ex: DEPTNO 가 첫 번째 칼럼이 된다.) 이때 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리한다.

[예제] SELECT * FROM EMP NATURAL JOIN DEPT;

[실행 결과] DEPTNO EMPNO ENAME JOB MGR HIREDATE SAL COMM DNAME LOC ----- ----- ----- -------- --- -------- ---- ---- --------- ------ 20 7369 SMITH CLERK 7902 1980-12-17 800 RESEARCH DALLAS 30 7499 ALLEN SALESMAN 7698 1981-02-20 1600 300 SALES CHICAGO 30 7521 WARD SALESMAN 7698 1981-02-22 1 250 500 SALES CHICAGO 20 7566 JONES MANAGER 7839 1981-04-02 2975 RESEARCH DALLAS 30 7654 MARTIN SALESMAN 7698 1981-09-28 1250 1400 S ALES CHICAGO 30 7698 BLAKE MANAGER 7839 1981-05-01 2850 SALES CHICAGO 10 7782 CLARK MANAGER 7839 1981-06-09 2450 ACCOUNTING NEW Y ORK 20 7788 SCOTT ANALYST 7566 1987-07-13 3000 RESEARCH DALLAS 10 7839 KING PRESIDENT 1981-11-17 5000 ACCOUNTING NEW YORK 30 7844 T URNER SALESMAN 7698 1981-09-08 1500 SALES CHICAGO 20 7876 ADAMS CLERK 7788 1987-07-13 1100 RESEARCH DALLAS 30 7900 JAMES CLERK 769 8 1981-12-03 950 0 SALES CHICAGO 20 7902 FORD ANALYST 7566 1981-12-03 3000 RESEARCH DALLAS 10 7934 MILLER CLERK 7782 1982-01-23 1300 AC COUNTING NEW YORK 14개의 행이 선택되었다.

[예제] 반면, INNER JOIN의 경우 첫 번째 테이블, 두 번째 테이블의 칼럼 순서대로 데이터가 출력된다. 이때 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리하지만, INNER JOIN은 별개의 칼럼으로 표시한다.

[예제] SELECT * FROM EMP INNER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

[실행 결과] EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO DEPTNO DNAME LOC ---- ----- ------ --- ------- --- ---- ----- ----- -------- ----- 7369 SMITH CLERK 7902 1980-12-17 800 20 20 RESEARCH DALLAS 7499 ALLEN SALESMAN 7698 1981-02-20 1600 300 30 30 SALES CHICAGO 7521 WARD SALESMAN 7698 19 81-02-22 1250 500 30 30 SALES CHICAGO 7566 JONES MANAGER 7839 1981-04-02 2975 20 20 RESEARCH DALLAS 7654 MARTIN SALESMAN 7698 1981-0 9-28 1250 1400 30 30 SALES CHICAGO 7698 BLAKE MANAGER 7839 1981-05-01 2850 30 30 SALES CHICAGO 7782 CLARK MANAGER 7839 1981-06-09 2450 10 10 ACCOUNTING NEW YORK 7788 SCOTT ANALYST 7566 1987-07-13 3000 20 20 RESEARCH DALLAS 7839 KING PRESIDENT 1981-11-17 5000 10 10 AC COUNTING NEW YORK 7844 TURNER SALESMAN 7698 1981-09-08 1500 0 30 30 SALES CHICAGO 7876 ADAMS CLERK 7788 1987-07-13 1100 20 20 RESEA RCH DALLAS 7900 JAMES CLERK 7698 1981-12-03 950 30 30 SALES CHICAGO 7902 FORD ANALYST 7566 1981-12-03 3000 20 20 RESEARCH DALLAS 7934 MILLER CLERK 7782 1982-01-23 1300 10 10 ACCOUNTING NEW YORK 14개의 행이 선택되었다.

[예제] NATURAL JOIN과 INNER JOIN의 차이를 자세히 설명하기 위해 DEPT_TEMP 테이블을 임시로 만든다.

[예제] Oracle CREATE TABLE DEPT_TEMP AS SELECT * FROM DEPT;

[예제] SQL Server SELECT * INTO DEPT_TEMP FROM DEPT;

[예제] UPDATE DEPT_TEMP SET DNAME = 'R&D' WHERE DNAME = 'RESEARCH'; UPDATE DEPT_TEMP SET DNAME = 'MARKETING' WHERE DNAME = 'SALE S'; SELECT * FROM DEPT_TEMP;

[실행 결과] DEPTNO DNAME LOC -------- ---------- --------- 10 ACCOUNTING NEW YORK 20 R&D DALLAS 30 MARKETING CHICAGO 40 OPERATIONS BOSTON 4개 의 행이 선택되었다.

부서번호 20과 30의 DNAME이 'R&D'와 'MARKETING'으로 변경된 것을 확인할 수 있다.

[예제] 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 NATURAL [INNER] JOIN으로 수행한다.

[예제] SELECT * FROM DEPT NATURAL INNER JOIN DEPT_TEMP; INNER는 DEFAULT 옵션으로 아래와 같이 생략 가능하므로 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다. SELECT * FROM DEPT NATURAL JOIN DEPT_TEMP;

[실행 결과] DEPTNO DNAME LOC ------ ---------- ---------- 10 ACCOUNTING NEW YORK 40 OPERATIONS BOSTON 2개의 행이 선택되었다.

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 데이터는 실행 결과에서 제외된 것을 알 수 있다.

[예제] 다음에는 같은 조건이지만 출력 칼럼에서 차이가 나는 일반적인 INNER JOIN을 수행한다.

[예제] SELECT * FROM DEPT JOIN DEPT_TEMP ON DEPT.DEPTNO = DEPT_TEMP.DEPTNO AND DEPT.DNAME = DEPT_TEMP.DNAME AND DEPT.LOC = DEPT_TEMP.LOC; 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다. SELECT * FROM DEPT, DEPT_TEMP WHERE DEPT.DEPTNO = DEPT_TEMP.DEPTNO AND DEPT.DNAME = DEPT_TEMP.DNAME AND DEPT.LOC = DEPT_TEMP.LOC;

[실행 결과] DEPTNO DNAME LOC DEPTNO DNAME LOC ------ ---------- -------- ------ ---------- ------ 10 ACCOUNTING NEW YORK 10 ACCOUNTING NEW YORK 40 OPERATIONS BOSTON 40 OPERATIONS BOSTON 2개의 행이 선택되었다.

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 차이가 나는 부분은 NATURAL JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리하지만, INNER JOIN의 경우는 2개의 칼럼으로 표시된다.

5. USING 조건절

NATURAL JOIN에서는 모든 일치되는 칼럼들에 대해 JOIN이 이루어지지만, FROM 절의 USING 조건절을 이용하면 같은 이름을 가진 칼럼들 중에서 원하는 칼럼에 대해서만 선택적으로 EQUI JOIN을 할 수가 있다. 다만, 이 기능은 SQL Server에서는 지원하지 않는다.

[예제] 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 DEPTNO 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

[예제] SELECT * FROM DEPT JOIN DEPT_TEMP USING (DEPTNO);

[실행 결과] DEPTNO DNAME LOC DNAME LOC ------ ---------- --------- ---------- --------- 10 ACCOUNTING NEW YORK ACCOUNTING NEW YORK 20 RESEARCH DALLAS R&D DALLAS 30 SALES CHICAGO MARKETING CHICAGO 40 OPERATIONS BOSTON OPERATIONS BOSTON 4개의 행이 선택되었다.

위 SQL의 '*' 와일드카드처럼 별도의 칼럼 순서를 지정하지 않으면 USING 조건절의 기준이 되는 칼럼이 다른 칼럼보다 먼저 출력된다. (ex: DEPTNO가 첫 번째 칼럼이 된다.) 이때 USING JOIN은 JOIN에 사용된 같은 이름의 칼럼을 하나로 처리한다.

[예제] USING 조건절을 이용한 EQUI JOIN에서도 NATURAL JOIN과 마찬가지로 JOIN 칼럼에 대해서는 ALIAS나 테이블 이름과 같은 접두사를 붙일 수 없다. (DEPT.DEPTNO → DEPTNO)

[예제] 잘못된 사례: SELECT DEPT.DEPTNO, DEPT.DNAME, DEPT.LOC, DEPT_TEMP.DNAME, DEPT_TEMP.LOC FROM DEPT JOIN DEPT_TEMP USING (DEPTNO); ERROR: USING 절의 열 부분은 식별자를 가질 수 없음 바른 사례: SELECT DEPTNO, DEPT.DNAME, DEPT.LOC, DEPT_TEMP.DNAME, DEPT_TEMP.LOC FROM DEPT JOIN DEPT_TEMP USING (DEPTNO);

[실행 결과] DEPTNO DNAME LOC DNAME LOC ------- --------- --------- ---------- -------- 10 ACCOUNTING NEW YORK ACCOUNTING NEW YORK 20 RESEARCH DALLAS R&D DALLAS 30 SALES CHICAGO MARKETING CHICAGO 40 OPERATIONS BOSTON OPERATIONS BOSTON 4개의 행이 선택되었다.

[예제] 이번에는 DEPT와 DEPT_TEMP 테이블의 일부 데이터 내용이 변경되었던 DNAME 칼럼을 조인 조건으로 [INNER] JOIN의 USING 조건절을 수행한다.

[예제] SELECT * FROM DEPT JOIN DEPT_TEMP USING (DNAME);

[실행 결과] DNAME DEPTNO LOC DEPTNO LOC ---------- ------ --------- ------- --------- ACCOUNTING 10 NEW YORK 10 NEW YORK OPERATIONS 40 BOSTON 40 BOSTON 2개의 행이 선택되었다.

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 그리고 USING에 사용된 DNAME이 첫 번째 칼럼으로 출력된 것과 함께, JOIN 조건에 참여하지 않은 DEPTNO와 LOC가 2개의 칼럼으로 표시된 것을 알 수 있다.

[예제] 이번에는 세 개의 칼럼명이 모두 같은 DEPT와 DEPT_TEMP 테이블을 LOC와 DEPTNO 2개 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

[예제] SELECT * FROM DEPT JOIN DEPT_TEMP USING (LOC, DEPTNO);

[실행 결과] LOC DEPTNO DNAME DNAME -------- ------ ---------- ---------- NEW YORK 10 ACCOUNTING ACCOUNTING DALLAS 20 RESEARCH R&D CHICAGO 30 SALES MARKETING BOSTON 40 OPERATIONS OPERATIONS 4개의 행이 선택되었다.

USING에 사용된 LOC, DEPTNO가 첫 번째, 두 번째 칼럼으로 출력되고, JOIN 조건에 참여하지 않은 DNAME 칼럼은 2개의 칼럼으로 표시된 것을 알 수 있다.

[예제] 이번에는 DEPTNO, DNAME 2개의 칼럼을 이용한 [INNER] JOIN의 USING 조건절로 수행한다.

[예제] SELECT * FROM DEPT JOIN DEPT_TEMP USING (DEPTNO, DNAME);

[실행 결과] DEPTNO DNAME LOC LOC ------ ---------- -------- -------- 10 ACCOUNTING NEW YORK NEW YORK 40 OPERATIONS BOSTON BOSTON 2개의 행이 선택되었다.

위 SQL의 경우 DNAME의 내용이 바뀐 부서번호 20, 30의 경우는 결과에서 제외된 것을 알 수 있다. 그리고 USING에 사용된 DEPTNO, DNAME이 첫 번째, 두 번째 칼럼으로 출력된 것과 함께, JOIN 조건에 참여하지 않은 LOC가 2개의 칼럼으로 표시된 것을 알 수 있다

6. ON 조건절

JOIN 서술부(ON 조건절)와 비 JOIN 서술부(WHERE 조건절)를 분리하여 이해가 쉬우며, 칼럼명이 다르더라도 JOIN 조건을 사용할 수 있는 장점이 있다.

[예제] 사원 테이블과 부서 테이블에서 사원 번호와 사원 이름, 소속부서 코드, 소속부서 이름을 출력한다.

[예제] SELECT E.EMPNO, E.ENAME, E.DEPTNO, D.DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO);

[실행 결과] EMPNO ENAME DEPTNO DNAME ----- ------- ------ ---------- 7369 SMITH 20 RESEARCH 7499 ALLEN 30 SALES 7521 WARD 30 SALES 7566 JONES 20 RESEARCH 7654 MARTIN 30 SALES 7698 BLAKE 30 SALES 7782 CLARK 10 ACCOUNTING 7788 SCOTT 20 RESEARCH 7839 KING 10 ACCOUNTING 7844 TU RNER 30 SALES 7876 ADAMS 20 RESEARCH 7900 JAMES 30 SALES 7902 FORD 20 RESEARCH 7934 MILLER 10 ACCOUNTING 14개의 행이 선택되었다.

NATURAL JOIN의 JOIN 조건은 기본적으로 같은 이름을 가진 모든 칼럼들에 대한 동등 조건이지만, 임의의 JOIN 조건을 지정하거나, 이름이 다른 칼럼명을 JOIN 조건으로 사용하거나, JOIN 칼럼을 명시하기 위해서는 ON 조건절을 사용한다. ON 조건절에 사용된 괄호는 옵션 사항이다. USING 조건절을 이용한 J OIN에서는 JOIN 칼럼에 대해서 ALIAS나 테이블 명과 같은 접두사를 사용하면 SYNTAX 에러가 발생하지만, 반대로 ON 조건절을 사용한 JOIN의 경우는 AL IAS나 테이블 명과 같은 접두사를 사용하여 SELECT에 사용되는 칼럼을 논리적으로 명확하게 지정해주어야 한다. (DEPTNO → E.DEPTNO) ON 조건절은 WHERE 절의 JOIN 조건과 같은 기능을 하면서도, 명시적으로 JOIN의 조건을 구분할 수 있으므로 가장 많이 사용될 것으로 예상된다. 다만, FROM 절에 테 이블이 많이 사용될 경우 다소 복잡하게 보여 가독성이 떨어지는 단점이 있다.

가. WHERE 절과의 혼용

[예제] ON 조건절과 WHERE 검색 조건은 충돌 없이 사용할 수 있다. 부서코드 30인 부서의 소속 사원 이름 및 소속 부서 코드, 부서 코드, 부서 이름을 찾아 본다.

[예제] SELECT E.ENAME, E.DEPTNO, D.DEPTNO, D.DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO) WHERE E.DEPTNO = 30;

[실행 결과] ENAME DEPTNO DEPTNO DNAME ------- ------ ------ ------ ALLEN 30 30 SALES WARD 30 30 SALES MARTIN 30 30 SALES BLAKE 30 30 SALES TURNE R 30 30 SALES JAMES 30 30 SALES 6개의 행이 선택되었다.

나. ON 조건절 + 데이터 검증 조건 추가

ON 조건절에 JOIN 조건 외에도 데이터 검색 조건을 추가할 수는 있으나, 검색 조건 목적인 경우는 WHERE 절을 사용할 것을 권고한다. (다만, 아우터 조인 에서 조인의 대상을 제한하기 위한 목적으로 사용되는 추가 조건의 경우는 ON 절에 표기되어야 한다.)

[예제] 매니저 사원번호가 7698번인 사원들의 이름 및 소속 부서 코드, 부서 이름을 찾아본다.

[예제] SELECT E.ENAME, E.MGR, D.DEPTNO, D.DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO AND E.MGR = 7698); 위 SQL과 아래 SQL은 같 은 결과를 얻을 수 있다. SELECT E.ENAME, E.MGR, D.DEPTNO, D.DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO) WHERE E.MGR = 7698;

[실행 결과] ENAME MGR DEPTNO DNAME ------- ---- ------ ------ ALLEN 7698 30 SALES WARD 7698 30 SALES MARTIN 7698 30 SALES TURNER 7698 30 SALES J AMES 7698 30 SALES 5개의 행이 선택되었다.

다. ON 조건절 예제

[예제] 팀과 스타디움 테이블을 스타디움ID로 JOIN하여 팀이름, 스타디움ID, 스타디움 이름을 찾아본다.

[예제] SELECT TEAM_NAME, TEAM.STADIUM_ID, STADIUM_NAME FROM TEAM JOIN STADIUM ON TEAM.STADIUM_ID = STADIUM.STADIUM_ID ORDER BY S TADIUM_ID;; 위 SQL은 STADIUM_ID라는 공통된 칼럼이 있기 때문에 아래처럼 USING 조건절로 구현할 수도 있다. SELECT TEAM_NAME, STADIUM_ID, STADIUM_N AME FROM TEAM JOIN STADIUM USING (STADIUM_ID) ORDER BY STADIUM_ID; 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SEL ECT TEAM_NAME, TEAM.STADIUM_ID, STADIUM_NAME FROM TEAM, STADIUM WHERE TEAM.STADIUM_ID = STADIUM.STADIUM_ID ORDER BY STADIUM_I D

[실행 결과] TEAM_NAME STADIUM_ID STADIUM_NAME ------------ --------- ------------ 광주상무 A02 광주월드컵경기장 강원FC A03 강릉종합경기장 제주유나이티드FC A04 제주월드컵경기장 대구FC A05 대구월드컵경기장 유나이티드 B01 인천월드컵경기장 일화천마 B02 성남종합운동장 삼성블루윙즈 B04 수원월드컵경기장 FC서울 B 05 서울월드컵경기장 아이파크 C02 부산아시아드경기장 울산현대 C04 울산문수경기장 경남FC C05 창원종합운동장 스틸러스 C06 포항스틸야드 드래곤즈 D01 광양전 용경기장 시티즌 D02 대전월드컵경기장 15개의 행이 선택되었다.

[예제] 팀과 스타디움 테이블을 팀ID로 JOIN하여 팀이름, 팀ID, 스타디움 이름을 찾아본다. STADIUM에는 팀ID가 HOMETEAM_ID라는 칼럼으로 표시되어 있다.

[예제] SELECT TEAM_NAME, TEAM_ID, STADIUM_NAME FROM TEAM JOIN STADIUM ON TEAM.TEAM_ID = STADIUM.HOMETEAM_ID ORDER BY TEAM_ID; 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SELECT TEAM_NAME, TEAM_ID, STADIUM_NAME FROM TEAM, STADIUM WHERE TEA M.TEAM_ID = STADIUM.HOMETEAM_ID ORDER BY TEAM_ID; 위 SQL은 TEAM_ID와 HOMETEAM_ID라는 다른 이름의 칼럼을 사용하기 때문에 USING 조건절을 사 용할 수는 없다.

[실행 결과] TEAM_NAME TEAM_ID STADIUM_NAME ---------- ------- ------------ 울산현대 K01 울산문수경기장 삼성블루윙즈 K02 수원월드컵경기장 스틸러스 K03 포항 스틸야드 유나이티드 K04 인천월드컵경기장 현대모터스 K05 전주월드컵경기장 아이파크 K06 부산아시아드경기장 드래곤즈 K07 광양전용경기장 일화천마 K08 성남종 합운동장 FC서울 K09 서울월드컵경기장 시티즌 K10 대전월드컵경기장 경남FC K11 창원종합운동장 광주상무 K12 광주월드컵경기장 강원FC K13 강릉종합경기장 제주 유나이티드FC K14 제주월드컵경기장 15개의 행이 선택되었다.

라. 다중 테이블 JOIN

[예제] 사원과 DEPT 테이블의 소속 부서명, DEPT_TEMP 테이블의 바뀐 부서명 정보를 출력한다.

[예제] SELECT E.EMPNO, D.DEPTNO, D.DNAME, T.DNAME New_DNAME FROM EMP E JOIN DEPT D ON (E.DEPTNO = D.DEPTNO) JOIN DEPT_TEMP T ON (E.D

EPTNO = T.DEPTNO); 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SELECT E.EMPNO, D.DEPTNO, D.DNAME, T.DNAME New_DNAME FROM EMP E, DEPT D, DEPT_TEMP T WHERE E.DEPTNO = D.DEPTNO AND E.DEPTNO = T.DEPTNO;

[실행 결과] EMPNO DEPTNO DNAME NEW_DNAME ------ ------ --------- ---------- 7369 20 RESEARCH R&D 7499 30 SALES MARKETING 7521 30 SALES MARKETING 7566 20 RESEARCH R&D 7654 30 SALES MARKETING 7698 30 SALES MARKETING 7782 10 ACCOUNTING ACCOUNTING 7788 20 RESEARCH R&D 7839 10 ACCOUNTING ACCOUNTING 7844 30 SALES MARKETING 7876 20 RESEARCH R&D 7900 30 SALES MARKETING 7902 20 RESEARCH R&D 7934 10 ACCOUNTING ACCOUNTING 14개의 행이 선택되었다.

[예제] GK 포지션의 선수별 연고지명, 팀명, 구장명을 출력한다.

[예제] SELECT P.PLAYER_NAME 선수명, P.POSITION 포지션, T.REGION_NAME 연고지명, T.TEAM_NAME 팀명, S.STADIUM_NAME 구장명 FROM PLAYER P JOIN TEAM T ON P.TEAM_ID = T.TEAM_ID JOIN STADIUM S ON T.STADIUM_ID = S.STADIUM_ID WHERE P.POSITION = 'GK' ORDER BY 선수명; 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SELECT P.PLAYER_NAME 선수명, P.POSITION 포지션, T.REGION_NAME 연고지명, T.TEAM_NAME 팀명, S.STADIUM_NAME 구장명 FROM PLAYER P, TEAM T, STADIUM S WHERE P.TEAM_ID = T.TEAM_ID AND T.STADIUM_ID = S.STADIUM_ID AND P.POSITION = 'GK' ORDER BY 선수명;

[실행 결과] 선수명 포지션 연고지명 팀명 구장명 ----- ---- ------ -------- ---------- 강성일 GK 대전 시티즌 대전월드컵경기장 권정혁 GK 울산 울산현대 울산문수경기장 권찬수 GK 성남 일화천마 성남종합운동장 김대희 GK 포항 스틸러스 포항스틸야드 김승준 GK 대전 시티즌 대전월드컵경기장 김용발 GK 전북 현대모터스 전주월드컵경기장 김운재 GK 수원 삼성블루윙즈 수원월드컵경기장 김정래 GK 전남 드래곤즈 광양전용경기장 김준호 GK 포항 스틸러스 포항스틸야드 김창민 GK 전북 현대모터스 전주월드컵경기장 김충호 GK 인천 유나이티드 인천월드컵경기장 남현우 GK 인천 유나이티드 인천월드컵경기장 박유석 GK 부산 아이파크 부산아시아드경기장 43개의 행이 선택되었다.

[예제] 홈팀이 3점 이상 차이로 승리한 경기의 경기장 이름, 경기 일정, 홈팀 이름과 원정팀 이름 정보를 출력한다.

[예제] SELECT ST.STADIUM_NAME, SC.STADIUM_ID, SCHE_DATE, HT.TEAM_NAME, AT.TEAM_NAME, HOME_SCORE, AWAY_SCORE FROM SCHEDULE SC JOIN STADIUM ST ON SC.STADIUM_ID = ST.STADIUM_ID JOIN TEAM HT ON SC.HOMETEAM_ID = HT.TEAM_ID JOIN TEAM AT ON SC.AWAYTEAM_ID = AT.TEAM_ID WHERE HOME_SCORE > = AWAY_SCORE +3; 위 SQL은 고전적인 방식인 WHERE 절의 INNER JOIN으로 구현할 수도 있다. SELECT ST.STADIUM_NAME, SC.STADIUM_ID, SCHE_DATE, HT.TEAM_NAME, AT.TEAM_NAME, HOME_SCORE, AWAY_SCORE FROM SCHEDULE SC, STADIUM ST, TEAM HT, TEAM AT WHERE HOME_SCORE> = AWAY_SCORE +3 AND SC.STADIUM_ID = ST.STADIUM_ID AND SC.HOMETEAM_ID = HT.TEAM_ID AND SC.AWAYTEAM_ID = AT.TEAM_ID; FROM 절에 4개의 테이블이 JOIN에 참여하였으며, HOME TEAM과 AWAY TEAM의 팀 이름을 구하기 위해 TEAM 테이블을 HT와 AT 두 개의 ALIAS로 구분하였다.

[실행 결과] STADIUM_NAME STADIUM_ID SCHE_DATE TEAM_NAME TEAM_NAME HOME_SCORE AWAY_SCORE ------------ --------- -------- --------- --------- --------- ---------- 서울월드컵경기장 B05 20120714 FC서울 삼성블루윙즈 3 0 부산아시아드경기장 C02 20120727 아이파크 시티즌 3 0 울산문수경기장 C04 20120803 울산현대 스틸러스 3 0 성남종합운동장 B02 20120317 일화천마 유나이티드 6 0 창원종합운동장 C05 20120427 경남FC 아이파크 5 2 5개의 행이 선택되었다.

7. CROSS JOIN

CROSS JOIN은 E.F.CODD 박사가 언급한 일반 집합 연산자의 PRODUCT의 개념으로 테이블 간 JOIN 조건이 없는 경우 생길 수 있는 모든 데이터의 조합을 말한다. 두 개의 테이블에 대한 CARTESIAN PRODUCT 또는 CROSS PRODUCT와 같은 표현으로, 결과는 양쪽 집합의 M*N 건의 데이터 조합이 발생한다. (아래 56건의 데이터는 EMP 14건 * DEPT 4건의 데이터 조합 건수이다.)

[예제] 사원 번호와 사원 이름, 소속부서 코드와 소속부서 이름을 찾아본다.

[예제] SELECT ENAME, DNAME FROM EMP CROSS JOIN DEPT ORDER BY ENAME;

[실행 결과] ENAME DNAME -------- --------- ADAMS SALES ADAMS RESEARCH ADAMS OPERATIONS ADAMS ACCOUNTING ALLEN OPERATIONS ALLEN RESEARCH ALLEN ACCOUNTING ALLEN SALES BLAKE SALES BLAKE OPERATIONS BLAKE RESEARCH BLAKE ACCOUNTING CLARK SALES CLARK RESEARCH CLARK OPERATIONS CLARK ACCOUNTING 56개의 행이 선택되었다.

[예제] NATURAL JOIN의 경우 WHERE 절에서 JOIN 조건을 추가할 수 없지만, CROSS JOIN의 경우 WHERE 절에 JOIN 조건을 추가할 수 있다. 그러나, 이 경우는 CROSS JOIN이 아니라 INNER JOIN과 같은 결과를 얻기 때문에 CROSS JOIN을 사용하는 의미가 없어지므로 권고하지 않는다.

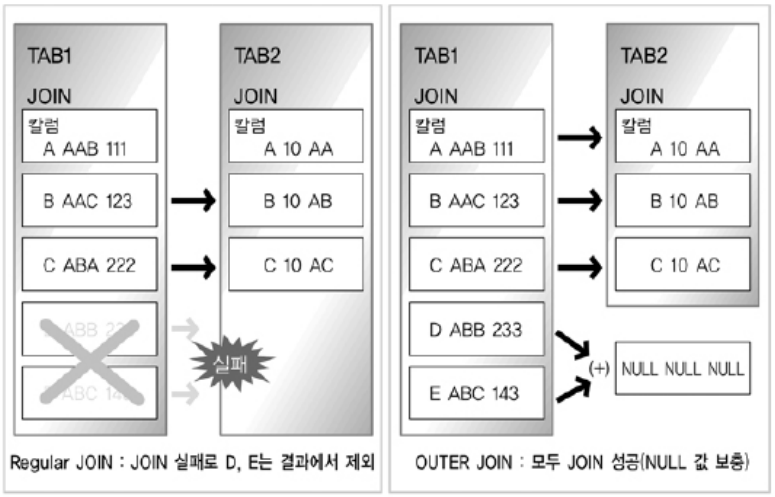
[예제] SELECT ENAME, DNAME FROM EMP CROSS JOIN DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO; 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다. SELECT ENAME, DNAME FROM EMP INNER JOIN DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO;

[실행 결과] ENAME DNAME ------- --------- SMITH RESEARCH ALLEN SALES WARD SALES JONES RESEARCH MARTIN SALES BLAKE SALES CLARK ACCOUNTING SCOTT RESEARCH KING ACCOUNTING TURNER SALES ADAMS RESEARCH JAMES SALES FORD RESEARCH MILLER ACCOUNTING 14개의 행이 선택되었다.

정상적인 데이터 모델이라면 CROSS PRODUCT가 필요한 경우는 많지 않지만, 간혹 튜닝이나 리포트를 작성하기 위해 고의적으로 사용하는 경우가 있을 수 있다. 그리고 데이터웨어하우스의 개별 DIMENSION(차원)을 FACT(사실) 칼럼과 JOIN하기 전에 모든 DIMENSION의 CROSS PRODUCT를 먼저 구할 때 유용하게 사용할 수 있다.

8. OUTER JOIN

INNER(내부) JOIN과 대비하여 OUTER(외부) JOIN이라고 불리며, JOIN 조건에서 동일한 값이 없는 행도 반환할 때 사용할 수 있다.



[그림 Ⅱ-2-3] OUTER JOIN 설명

[그림 Ⅱ-2-3]은 TAB1 테이블이 TAB2 테이블을 JOIN 하되, TAB2의 JOIN 데이터가 있는 경우는 TAB2의 데이터를 함께 출력하고, TAB2의 JOIN 데이터가 없는 경우에도 TAB1의 모든 데이터를 표시하고 싶은 경우이다. TAB1의 모든 값에 대해 TAB2의 데이터가 반드시 존재한다는 보장이 없는 경우 OUTER JOIN을 사용하여 해결이 가능하다. 과거 OUTER JOIN을 위해 Oracle은 JOIN 칼럼 뒤에 '(+)'를 표시하였고, Sybase는 비교 연산자의 앞이나 뒤에 '(+)'를 표시했었는데, JOIN 조건과 WHERE 절 검색 조건이 불명확한 단점, IN이나 OR 연산자 사용시 에러 발생, '(+)' 표시가 누락된 칼럼 존재시 OUTER JOIN 오류 발생, FULL OUTER JOIN 미지원 등 불편함이 많았다. STANDARD JOIN을 사용함으로써 OUTER JOIN의 많은 문제점을 해결할 수 있고, 대부분의 관계형 DBMS 간에 호환성을 확보할 수 있으므로 명시적인 OUTER JOIN을 사용할 것을 적극적으로 권장한다. 추가로 OUTER JOIN 역시 JOIN 조건을 FROM 절에서 정의하겠다는 표시이므로 USING 조건절이나 ON 조건절을 필수적으로 사용해야 한다. 그리고, LEFT/RIGHT OUTER JOIN의 경우에는 기준이 되는 테이블이 조인 수행시 무조건 드라이빙 테이블이 된다. 옵티마이저는 이 원칙에 위배되는 다른 실행계획을 고려하지 않는다.

가. LEFT OUTER JOIN

조인 수행시 먼저 표기된 좌측 테이블에 해당하는 데이터를 먼저 읽은 후, 나중 표기된 우측 테이블에서 JOIN 대상 데이터를 읽어 온다. 즉, Table A와 B가 있을 때(Table 'A'가 기준이 됨), A와 B를 비교해서 B의 JOIN 칼럼에서 같은 값이 있을 때 그 해당 데이터를 가져오고, B의 JOIN 칼럼에서 같은 값이 없는 경우에는 B 테이블에서 가져오는 칼럼들은 NULL 값으로 채운다. 그리고 LEFT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

[예제] STADIUM에 등록된 운동장 중에는 홈팀이 없는 경기장도 있다. STADIUM과 TEAM을 JOIN 하되 홈팀이 없는 경기장의 정보도 같이 출력하도록 한다.

[예제] SELECT STADIUM_NAME, STADIUM.STADIUM_ID, SEAT_COUNT, HOMETEAM_ID, TEAM_NAME FROM STADIUM LEFT OUTER JOIN TEAM ON STADIUM.HOMETEAM_ID = TEAM.TEAM_ID ORDER BY HOMETEAM_ID; OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다. SELECT STADIUM_NAME, STADIUM.STADIUM_ID, SEAT_COUNT, HOMETEAM_ID, TEAM_NAME FROM STADIUM LEFT JOIN TEAM ON STADIUM.HOMETEAM_ID = TEAM.TEAM_ID ORDER BY HOMETEAM_ID;

[실행 결과] STADIUM_NAME STADIUM_ID SEAT_COUNT HOMETEAM_ID TEAM_NAME ---------- --------- ---------- ---------- ---------- 울산문수경기장 C04 46102 K01 울산현대 수원월드컵경기장 B04 50000 K02 삼성블루윙즈 포항스틸야드 C06 25000 K03 스틸러스 인천월드컵경기장 B01 35000 K04 유나이티드 전주월드컵경기장 D03 28000 K05 현대모터스 부산아시아드경기장 C02 30000 K06 아이파크 광양전용경기장 D01 20009 K07 드래곤즈 성남종합운동장 B02 27000 K08 일화천마 서울월드컵경기장 B05 66806 K09 FC서울 대전월드컵경기장 D02 41000 K10 시티즌 창원종합운동장 C05 27085 K11 경남FC 광주월드컵경기장 A02 40245 K12 광주상무 강릉종합경기장 A03 33000 K13 강원FC 제주월드컵경기장 A04 42256 K14 제주유나이티드FC 대구월드컵경기장 A05 66422 K15 대구FC 안양경기장 F05 20000 마산경기장 F04 20000 일산경기장 F03 20000 부산시민경기장 F02 30000 대구시민경기장 F01 30000 20개의 행이 선택되었다.

INNER JOIN이라면 홈팀이 배정된 15개의 경기장만 출력 되었겠지만, LEFT OUTER JOIN을 사용하였기 때문에 홈팀이 없는 대구시민경기장, 부산시민경기장, 일산경기장, 마산경기장, 안양경기장의 정보까지 추가로 출력되었다.

나. RIGHT OUTER JOIN

조인 수행시 LEFT JOIN과 반대로 우측 테이블이 기준이 되어 결과를 생성한다. 즉, TABLE A와 B가 있을 때(TABLE 'B'가 기준이 됨), A와 B를 비교해서 A의 JOIN 칼럼에서 같은 값이 있을 때 그 해당 데이터를 가져오고, A의 JOIN 칼럼에서 같은 값이 없는 경우에는 A 테이블에서 가져오는 칼럼들은 NULL 값으로 채운다. 그리고 RIGHT JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

[예제] DEPT에 등록된 부서 중에는 사원이 없는 부서도 있다. DEPT와 EMP를 조인하되 사원이 없는 부서 정보도 같이 출력하도록 한다.

[예제] SELECT E.ENAME, D.DEPTNO, D.DNAME FROM EMP E RIGHT OUTER JOIN DEPT D ON E.DEPTNO = D.DEPTNO; OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다. SELECT E.ENAME, D.DEPTNO, D.DNAME, D.LOC FROM EMP E RIGHT JOIN DEPT D ON E.DEPTNO = D.DEPTNO;

[실행 결과] ENAME DEPTNO DNAME LOC ----- ------ ---------- -------- CLARK 10 ACCOUNTING NEW YORK KING 10 ACCOUNTING NEW YORK MILLER 10 ACCOUNTING NEW YORK JONES 20 RESEARCH DALLAS FORD 20 RESEARCH DALLAS ADAMS 20 RESEARCH DALLAS SMITH 20 RESEARCH DALLAS SCOTT 20 RESEARCH DALLAS WARD 30 SALES CHICAGO TURNER 30 SALES CHICAGO ALLEN 30 SALES CHICAGO JAMES 30 SALES CHICAGO BLAKE 30 SALES CHICAGO MARTIN 30 SALES CHICAGO 40 OPERATIONS BOSTON 15개의 행이 선택되었다.

INNER JOIN이라면 사원 정보와 함께 사원이 배정된 3개의 부서 정보와 14명의 사원 정보만 출력 되었겠지만, RIGHT OUTER JOIN을 사용하였기 때문에 사원이 배정되지 않은 부서번호 40의 OPERATIONS 부서의 LOC 정보까지 출력되었다.

다. FULL OUTER JOIN

조인 수행시 좌측, 우측 테이블의 모든 데이터를 읽어 JOIN하여 결과를 생성한다. 즉, TABLE A와 B가 있을 때(TABLE 'A', 'B' 모두 기준이 됨), RIGHT OUTER JOIN과 LEFT OUTER JOIN의 결과를 합집합으로 처리한 결과와 동일하다. 단, UNION ALL이 아닌 UNION 기능과 같으므로 중복되는 데이터는 삭제한다.

(UNION ALL과 UNION에 대해서는 다음 절에서 설명하도록 한다.) 그리고 FULL JOIN으로 OUTER 키워드를 생략해서 사용할 수 있다.

[예제] DEPT 테이블과 DEPT_TEMP 테이블의 FULL OUTER JOIN 사례를 만들기 위해 DEPT_TEMP의 DEPTNO를 수정한다. 결과적으로 DEPT_TEMP 테이블의 새로운 DEPTNO 데이터는 DETP 테이블의 DEPTNO와 2건은 동일하고 2건은 새로운 DEPTNO가 생성된다.

[예제] UPDATE DEPT_TEMP SET DEPTNO = DEPTNO + 20; SELECT * FROM DEPT_TEMP;

[실행 결과] DEPTNO DNAME LOC ------ ---------- ---------- 30 ACCOUNTING NEW YORK 40 R&D DALLAS 50 MARKETING CHICAGO 60 OPERATIONS BOSTON 4개의 행이 선택되었다.

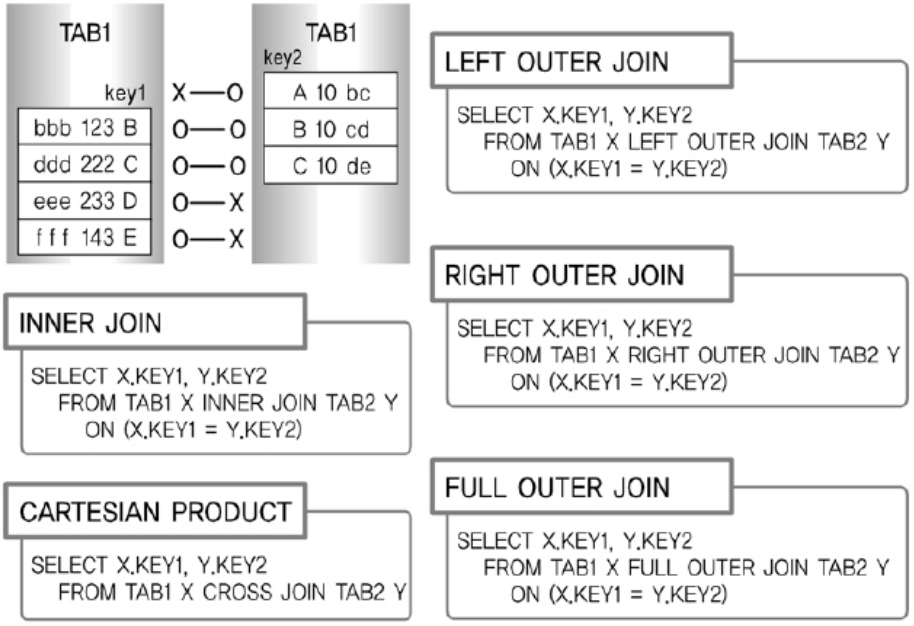
[예제] DEPTNO 기준으로 DEPT와 DEPT_TEMP 데이터를 FULL OUTER JOIN으로 출력한다. 예제에 사용된 UNION(중복 데이터는 제거됨)은 다음 절에서 설명하도록 한다.

[예제] SELECT * FROM DEPT FULL OUTER JOIN DEPT_TEMP ON DEPT.DEPTNO = DEPT_TEMP.DEPTNO; OUTER는 생략 가능한 키워드이므로 아래 SQL은 같은 결과를 얻을 수 있다. SELECT * FROM DEPT FULL JOIN DEPT_TEMP ON DEPT.DEPTNO = DEPT_TEMP.DEPTNO; 위 SQL과 아래 SQL은 같은 결과를 얻을 수 있다. SELECT L.DEPTNO, L.DNAME, L.LOC, R.DEPTNO, R.DNAME, R.LOC FROM DEPT L LEFT OUTER JOIN DEPT_TEMP R ON L.DEPTNO = R.DEPTNO UNION SELECT L.DEPTNO, L.DNAME, L.LOC, R.DEPTNO, R.DNAME, R.LOC FROM DEPT L RIGHT OUTER JOIN DEPT_TEMP R ON L.DEPTNO = R.DEPTNO;

[실행 결과] DEPTNO DNAME LOC DEPTNO DNAME LOC ------ ---------- -------- ------ ---------- ------ 10 ACCOUNTING NEW YORK 20 RESEARCH DALLAS 30 SALES CHICAGO 30 ACCOUNTING NEW YORK 40 OPERATIONS BOSTON 40 R&D DALLAS 50 MARKETING CHICAGO 60 OPERATIONS BOSTON 6개의 행이 선택되었다.

INNER JOIN이라면 부서번호가 동일한 30, 40 부서의 2개 정보만 출력되었겠지만, FULL OUTER JOIN을 사용하였기 때문에 DEPT 테이블에만 있는 부서번호 10, 20의부서와 DEPT_TEMP 테이블에만 있는 부서번호 50, 60의 부서 정보까지 같이 출력되었다.

9. INNER vs OUTER vs CROSS JOIN 비교



[그림 Ⅱ-2-4] INNER vs OUTER vs CROSS JOIN 문장 비교

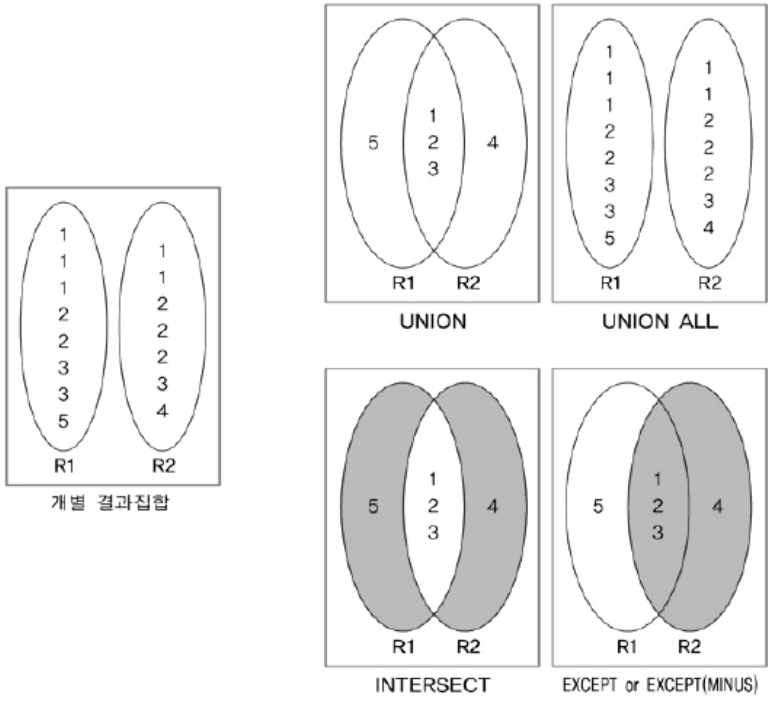
첫 번째, INNER JOIN의 결과는 다음과 같다. 양쪽 테이블에 모두 존재하는 키 값이 B-B, C-C 인 2건이 출력된다. 두 번째, LEFT OUTER JOIN의 결과는 다음과 같다. TAB1을 기준으로 키 값 조합이 B-B, C-C, D-NULL, E-NULL 인 4건이 출력된다. 세 번째, RIGHT OUTER JOIN의 결과는 다음과 같다. TAB2를 기준으로 키 값 조합이 NULL-A, B-B, C-C 인 3건이 출력된다. 네 번째, FULL OUTER JOIN의 결과는 다음과 같다. 양쪽 테이블을 기준으로 키 값 조합이 NULL-A, B-B, C-C, D-NULL, E-NULL 인 5건이 출력된다. 다섯 번째, CROSS JOIN의 결과는 다음과 같다. JOIN 가능한 모든 경우의 수를 표시하지만 단, OUTER JOIN은 제외한다. 양쪽 테이블 TAB1과 TAB2의 데이터를 곱한 개수인 4 * 3 = 12건이 추출됨 키 값 조합이 B-A, B-B, B-C, C-A, C-B, C-C, D-A, D-B, D-C, E-A, E-B, E-C 인 12건이 출력된다.

두 개 이상의 테이블에서 조인을 사용하지 않고 연관된 데이터를 조회하는 방법 중에 또 다른 방법이 있는데 그 방법이 바로 집합 연산자(Set Operator)를 사용하는 방법이다. 기존의 조인에서는 FROM 절에 검색하고자 하는 테이블을 나열하고, WHERE 절에 조인 조건을 기술하여 원하는 데이터를 조회할 수 있었다. 하지만 집합 연산자는 여러 개의 질의의 결과를 연결하여 하나로 결합하는 방식을 사용한다. 즉, 집합 연산자는 2개 이상의 질의 결과를 하나의 결과로 만들어 준다. 일반적으로 집합 연산자를 사용하는 상황은 서로 다른 테이블에서 유사한 형태의 결과를 반환하는 것을 하나의 결과로 합치고자 할 때와 동일 테이블에서 서로 다른 질의를 수행하여 결과를 합치고자 할 때 사용할 수 있다. 이외에도 튜닝관점에서 실행계획을 분리하고자 하는 목적으로도 사용할 수 있다. 집합 연산자를 사용하기 위해서는 다음 제약조건을 만족해야 한다. SELECT 절의 칼럼 수가 동일하고 SELECT 절의 동일 위치에 존재하는 칼럼의 데이터 타입이 상호 호환 가능(반드시 동일한 데이터 타입일 필요는 없음)해야 한다. 그렇지 않으면 데이터베이스가 오류를 반환한다.

[표 Ⅱ-2-1] 집합 연산자의 종류

집합 연산자	연산자의 의미
UNION	여러 개의 SQL문의 결과에 대한 합집합으로 결과에서 모든 중복된 행은 하나의 행으로 만든다.
UNION ALL	여러 개의 SQL문의 결과에 대한 합집합으로 중복된 행도 그대로 결과로 표시된다. 즉, 단순히 결과만 합쳐놓은 것이다. 일반적으로 여러 질의 결과가 상호 배타적인(Exclusive)일 때 많이 사용한다. 개별 SQL문의 결과가 서로 중복되지 않는 경우, UNION과 결과가 동일하다. (결과의 정렬 순서에는 차이가 있을 수 있음)
INTERSECT	여러 개의 SQL문의 결과에 대한 교집합이다. 중복된 행은 하나의 행으로 만든다.
EXCEPT	앞의 SQL문의 결과에서 뒤의 SQL문의 결과에 대한 차집합이다. 중복된 행은 하나의 행으로 만든다. (일부 데이터베이스는 MINUS를 사용함)

집합 연산자는 개별 SQL문의 결과 집합에 대해 합집합(UNION/UNION ALL), 교집합(INTERSECT), 차집합(EXCEPT)으로 집합간의 관계를 가지고 작업을 한다.



[그림 Ⅱ-2-5] 집합 연산자의 연산

집합 연산자를 가지고 연산한 결과는 [그림 Ⅱ-2-5]와 같다. [그림 Ⅱ-2-5]의 왼쪽에 존재하는 R1, R2는 각각의 SQL문을 실행해서 생성된 개별 결과 집합을 의미한다. [그림 Ⅱ-2-5]에서 보면 알 수 있듯이 UNION ALL을 제외한 다른 집합 연산자에서는 SQL문의 결과 집합에서 먼저 중복된 건을 배제하는 작업을 수행한 후에 집합 연산을 적용한다(논리적인 관점의 처리임). UNION 연산에서 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되고, 이것의 합집합(R1 ∪ R2)의 결과는 {1, 2, 3, 4, 5}이다. UNION ALL 연산은 중복에 대한 배제 없이 2개의 결과 집합을 단순히 합친 것과 동일한 결과이다. UNION ALL의 결과는 {1, 1, 1, 2, 2, 3, 3, 5, 1, 1, 2, 2, 3, 4}이다. INTERSECT 연산에서 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되어, 이것의 교집합(R1 ∩ R2)의 결과는 {1, 2, 3}이다. EXCEPT 연산에서는 R1 = {1, 2, 3, 5}, R2 = {1, 2, 3, 4}가 되고, 이것의 차집합(R1 R2)의 결과는 {5}이다. EXCEPT 연산에서는 순서가 중요하다. 만약 순서가 바뀌어서 R2 R1의 차집합이었다면 결과는 {4}가 된다. 집합 연산자를 사용하여 만들어지는 SQL문의 형태는 다음과 같다.

SELECT 칼럼명1, 칼럼명2, ... FROM 테이블명1 [WHERE 조건식] [[GROUP BY 칼럼(Column)이나 표현식 [HAVING 그룹조건식]] 집합 연산자 SELECT 칼럼명1, 칼럼명2, ... FROM 테이블명2 [WHERE 조건식] [[GROUP BY 칼럼(Column)이나 표현식 [HAVING 그룹조건식]] [ORDER BY 1, 2 [ASC또는 DESC] ; SELECT PLAYER_NAME 선수명, BACK_NO 백넘버 FROM PLAYER WHERE TEAM_ID = 'K02' UNION SELECT PLAYER_NAME 선수명, BACK_NO 백넘버 FROM PLAYER WHERE TEAM_ID = 'K07' ORDER BY 1;

집합 연산자는 사용상의 제약조건을 만족한다면 어떤 형태의 SELECT문이라도 이용할 수 있다. 집합 연산자는 여러 개의 SELECT문을 연결하는 것에 지나지 않는다. ORDER BY는 집합 연산을 적용한 최종 결과에 대한 정렬 처리이므로 가장 마지막 줄에 한번만 기술한다. 아래 질문에 대해 집합 연산자를 사용하여 처리하는 방법을 알아보자.

[집합 연산자를 연습하기 위한 질문]

1) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과전남드레곤즈팀인 선수들에 대한 내용을 모두보고 싶다. 2) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과 포지션이 골키퍼(GK)인 선수들을 모두 보고 싶다. 3) K-리그 소속 선수들에 대한 정보 중에서 포지션별 평균키와 팀별 평균키를 알고 싶다. 4) K-리그 소속 선수를 중에서 소속이 삼성블루윙즈팀이면서 포지션이 미드필더(MF)가 아닌 선수들의 정보를 보고 싶다. 5) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀이면서 포지션이 골키퍼(GK)인 선수들의 정보를 보고 싶다.

SQL문을 작성하기에 전에 [집합 연산자를 연습하기 위한 질문]을 다음과 같이 집합 연산자를 사용 형태로 해석할 수 있다.

[질문을 집합 연산의 개념으로 해석한 결과]

1) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 소속이 전남드레곤즈팀인 선수들의 집합의 합집합 2) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 합집합 3) K-리그 소속 선수 중 포지션별 평균키에 대한 집합과K-리그 소속 선수 중 팀별 평균키에 대한 집합의 합집합 4) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 미드필더(MF))인 선수들의 집합의 차집합 5) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 교집합

위의 결과를 집합 연산자를 사용하여 SQL문을 작성하고 그 결과를 확인해 보도록 하자. 먼저 첫 번째 질문에 대한 SQL문을 작성하고 실행해 보자.

[질문1] 1) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과전남드레곤즈팀인 선수들에 대한 내용을 모두보고 싶다. 1) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 소속이 전남드레곤즈팀인 선수들의 집합의 합집합

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' UNION SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K07'

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ---- ---- ---- -- K02 가비 MF 10 177 K02 강대희 MF 26 174 K02 고종수 MF 22 176 K02 고창현 MF 8 170 K02 김강진 DF 43 181 K07 강철 DF 3 178 K07 김반 MF 14 174 K07 김영수 MF 30 175 K07 김정래 GK 33 185 K07 김창원 DF 5 183 100개의 행이 선택되었다.

첫 번째 질문에 대한 SQL문에서 삼성 블루윙즈팀인 선수들과 전남 드레곤즈팀의 선수들의 합집합이라는 것은 WHERE 절에 IN 또는 OR 연산자로도 변환이 가능하다. 다만 IN 또는 OR 연산자를 사용할 경우에는 결과의 표시 순서가 달라질 수 있다. 집합이라는 관점에서는 결과가 표시되는 순서가 틀렸다고 두 집합이 서로 다르다고 말할 수 없다. 만약, 결과의 동일한 표시 순서를 원한다면 ORDER BY절을 사용해서 명시적으로 정렬 순서를 정의하는 것이 바람직하다. 첫 번째 질문에 대한 SQL문을 IN 또는 OR 연산자를 사용한 SQL문으로 변경해 보고 결과의 표시 순서가 다름을 확인해 보자.

[예제] (비교) SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' OR TEAM_ID = 'K07'; SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID IN ('K02', 'K07');

[실행 결과] 팀 선수명 포지션 백넘버 키 ---- ----- ---- ----- --- K07 김회택 TM K07 서현옥 TC K07 정상호 TC K07 최철우 TC K07 정영광 GK 41 185 K02 정호 TM K02 왕선재 TC K02 코샤 TC K02 윤성효 TC K02 정광수 GK 41 182 100개의 행이 선택되다.

두 번째 질문에 대해 SQL문을 작성하고 결과를 확인해 보자.

[질문2] 2) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀인 선수들과 포지션이 골키퍼(GK)인 선수들을 모두 보고 싶다. 2) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 합집합

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' UNION SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'GK';

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ----- ----- ---- -- K01 권정혁 GK 1 195 K01 서동명 GK 21 196 K01 양지원 GK 45 181 K01 이무림 GK 31 185 K01 최창주 GK 40 187 K02 가비 MF 10 177 K02 강대희 MF 26 174 K02 고종수 MF 22 176 K02 고창현 MF 8 170 K02 김강진 DF 43 181 88개의 행이 선택되었다.

두 번째 질문에 대한 실행 결과는 첫 번째 실행 결과와 비교해 보면 집합의 대상만 차이가 날뿐 다른 점은 없다. 마찬가지로 두 번째 질문에 대해 OR 연산자를 사용한 SQL문으로 변경하면 다음과 같다. 여기서는 서로 다른 칼럼에 조건을 사용했기 때문에 IN 연산자를 사용할 수 없다.

[예제] (비교) SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' OR POSITION = 'GK';

만약, 두 번째 질문에 대한 SQL문에서 UNION이라는 집합 연산자 대신에 UNION ALL이라는 집합 연산자를 사용하면 어떻게 될지 한번 수행해 보자.

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' UNION ALL SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'GK';

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ----- ---- ---- --- K02 정호 TM K02 왕선재 TC K02 코샤 TC K02 윤성효 TC K02 정광수 GK 41 182 K04 남현우 GK 31 180 K04 김충호 GK 60 185 K04 이현 GK 1 192 K04 한동진 GK 21 183 K10 강성일 GK 30 182 92개의 행이 선택되었다.

수행 결과에서 알 수 있듯이 결과 건수가 UNION은 88건이었으나 UNION ALL은 92건으로 결과 건수가 늘어났다. 두 SQL문의 결과가 서로 다르다. 결과가 다

른 이유는 UNION은 결과에서 중복이 존재할 경우 중복을 제외시키지만 UNION ALL은 각각의 질의 결과를 단순히 결합시켜 줄 뿐 중복된 결과를 제외시키지 않기 때문이다. 이와 같이 결과 집합에 중복이 존재하면 UNION과 UNION ALL의 결과는 달라진다. UNION ALL에서 중복된 결과들을 확인해 보고자 할 때는 ORDER BY절을 사용하면 용이하다. 아래 SQL문을 통해 중복된 결과를 확인해 보자.

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' UNION ALL S ELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'GK' ORDER BY 1, 2, 3, 4, 5;

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ---- ---- ---- --- K02 김운재 GK 1 182 K02 김운재 GK 1 182 K02 정광수 GK 41 182 K02 정광수 GK 41 182 K02 조범철 GK 21 185 K02 조범철 GK 21 185 K02 최호진 GK 31 190 K02 최호진 GK 31 190 92개의 행이 선택되었다.

결과에서 삼성블루윙즈팀(K02)에서 포지션이 골키퍼(GK)인 사람이 중복 표시되째 질문에 대한 SQL문을 작성하고 결과를 확인해 보자.

[질문3] 3) K-리그 소속 선수들에 대한 정보 중에서 포지션별 평균키와 팀별 평균키를 알고 싶다. 3) K-리그 소속 선수 중 포지션별 평균키에 대한 집합과K-리그 소속 선수 중 팀별 평균키에 대한 집합의 합집합

[예제] SELECT 'P' 구분코드, POSITION 포지션, AVG(HEIGHT) 평균키 FROM PLAYER GROUP BY POSITION UNION SELECT 'T' 구분코드, TEAM_ID 팀명, AVG(HEIG HT) 평균키 FROM PLAYER GROUP BY TEAM_ID ORDER BY 1;

[실행 결과] 구분코드 포지션 평균키 ------ ----- -------- P DF 180.409 P FW 179.91 P GK 186.256 P MF 176.309 P TC 178.833 T K01 180.089 T K02 179.067 T K03 179.9 11 T K04 180.511 T K05 180.422 23개의 행이 선택되었다.

세 번째 질문에서는 평균키에 대한 값들의 합집합을 구하는 것이다. 합집합을 구하기 위해 SQL문에서 그룹함수를 사용했다. 그룹함수도 집합 연산자에서 사용이 가능하다는 것을 알 수 있다. 또한 실제로 테이블에는 존재하지 않지만 결과 행을 구분하기 위해 SELECT 절에 칼럼('구분코드')을 추가할 수 있다는 것을 알 수 있다. 이와 같이 목적을 위해 SELECT 절에 임의의 칼럼을 추가하는 것은 다른 모든 SQL문에서 적용 가능하다. 집합 연산자의 결과를 표시할 때 HE ADING 부분은 첫 번째 SQL문에서 사용된 HEADING이 적용된다는 것을 알 수 있다. SQL문에서 첫 번째 SELECT 절에서는 '포지션' HEADING을 사용하였고 두 번째 SELECT 절에서는 '팀명' HEADING을 사용하였다. 그러나 결과에는 '포지션' HEADING으로 표시되었다.

네 번째 질문인 삼성 블루윙즈팀인 집합과 포지션이 미드필더(MF)인 선수들의 차집합에 대한 SQL문을 작성하고 결과를 확인해 보자.

[질문4] 4) K-리그 소속 선수를 중에서 소속이 삼성블루윙즈팀이면서 포지션이 미드필더(MF)가 선수들의 정보를 보고 싶다. 4) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 미드필더(MF))인 선수들의 집합의 차집합

[예제] Oracle SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' MINUS SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'MF' ORDER BY 1, 2, 3, 4, 5;

SQL Server에서는 MINUS대신 EXCEPT를 사용할 수 있다.

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ------- ----- ---- -- K02 김강진 DF 43 181 K02 김관희 FW 39 180 K02 김만근 FW 34 177 K02 김병국 DF 2 183 K02 김병근 DF 3 175 K02 왕선재 TC K02 윤성효 TC K02 윤화평 FW 42 182 K02 이성용 DF 20 173 K02 정광수 GK 41 182 31개의 행이 선택되었다.

차집합은 앞의 집합의 결과에서 뒤의 집합의 결과를 빼는 것이다. 이번 SQL문은 삼성블루윙즈팀의 선수들 중에서 포지션이 미드필더(MF)인 선수들의 정보를 빼는 것이다. 해당 SQL문은 다른 형태의 SQL문으로 변경 가능하다. EXCEPT 연산자의 앞에 오는 SQL문의 조건은 만족하고 뒤에 오는 SQL문의 조건은 만족하지 않는 SQL문과 동일한 결과를 얻을 수 있다. 그러므로 EXCEPT 연산자를 사용하지 않고 논리 연산자를 이용하여 동일한 결과의 SQL문을 작성할 수 있다. SQL문을 실행하여 결과가 동일한지 직접 확인해 보자.

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' AND POSITIO N <> 'MF' ORDER BY 1, 2, 3, 4, 5;

MINUS 연산자는 NOT EXISTS 또는 NOT IN 서브쿼리를 이용한 SQL문으로도 변경 가능하다. (NOT EXISTS와 NOT IN에 대한 설명은 제5절 서브쿼리에서 참고)

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER X WHERE X.TEAM_ID = 'K02' AND NOT EXISTS (SELECT 1 FROM PLAYER Y WHERE Y.PLAYER_ID = X.PLAYER_ID AND POSITION = 'MF') ORDER BY 1, 2, 3, 4, 5; SELECT TEAM_ID 팀코드, PLAYE R_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' AND PLAYER_ID NOT IN (SELECT PLAYER_ID FRO M PLAYER WHERE POSITION = 'MF') ORDER BY 1, 2, 3, 4, 5;

이제 마지막으로 삼성블루윙즈팀이면서 포지션이 골키퍼인 선수들인 교집합을 얻기 위한 SQL문을 작성해 보자.

[질문 5] 5) K-리그 소속 선수들 중에서 소속이 삼성블루윙즈팀이면서 포지션이 골키퍼(GK)인 선수들의 정보를 보고 싶다. 5) K-리그 소속 선수 중 소속이 삼성블루윙즈팀인 선수들의 집합과K-리그 소속 선수 중 포지션이 골키퍼(GK)인 선수들의 집합의 교집합

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' INTERSECT S ELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE POSITION = 'GK' ORDER BY 1, 2, 3, 4, 5;

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ---- ------ ----- ---- -- K02 김운재 GK 1 182 K02 정광수 GK 41 182 K02 조범철 GK 21 185 K02 최호진 GK 31 190 4개의 행이 선택되었다.

교집합의 결과는 소속이 삼성 블루윙즈팀인 선수의 집합이면서 포지션이 골키퍼인 집합인 두 개의 조건을 만족하는 집합이다. 이것은 INTERSECT 연산자의 앞에 오는 SQL문의 조건은 만족하면서 뒤의 SQL문의 조건을 만족하는 것과 동일한 결과를 얻을 수 있다. 다음과 같이 INTERSECT 연산자를 사용하지 않고도 논리 연산자만으로 결과가 동일한 SQL문을 작성할 수 있다. SQL문을 실행하여 결과가 동일한지 직접 확인해 보자.

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' AND POSITIO
N = 'GK' ORDER BY 1, 2, 3, 4, 5;

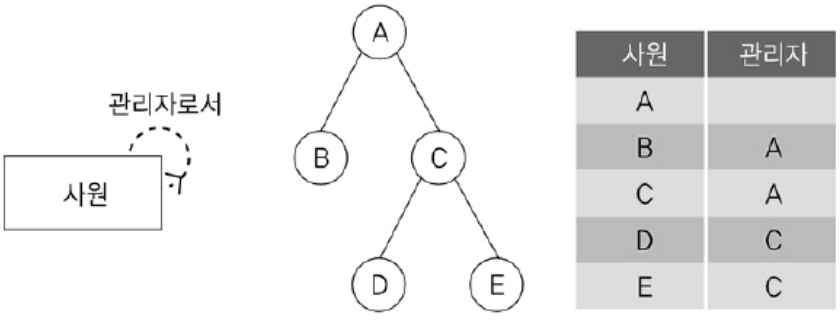
INTERSECT 연산자는 EXISTS 또는 IN 서브쿼리를 이용한 SQL문으로 변경 가능하다. (EXISTS와 IN에 대한 설명은 제5절 서브쿼리에서 참고)

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER X WHERE X.TEAM_ID = 'K02' AND EXIS
TS (SELECT 1 FROM PLAYER Y WHERE Y.PLAYER_ID = X.PLAYER_ID AND Y.POSITION = 'GK') ORDER BY 1, 2, 3, 4, 5;

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE TEAM_ID = 'K02' AND PLAYE
R_ID IN (SELECT PLAYER_ID FROM PLAYER WHERE POSITION = 'GK') ORDER BY 1, 2, 3, 4, 5;

1. 계층형 질의

테이블에 계층형 데이터가 존재하는 경우 데이터를 조회하기 위해서 계층형 질의(Hierarchical Query)를 사용한다. 계층형 데이터란 동일 테이블에 계층적으로 상위와 하위 데이터가 포함된 데이터를 말한다. 예를 들어, 사원 테이블에서는 사원들 사이에 상위 사원(관리자)과 하위 사원 관계가 존재하고 조직 테이블에서는 조직들 사이에 상위 조직과 하위 조직 관계가 존재한다. 엔터티를 순환관계 데이터 모델로 설계할 경우 계층형 데이터가 발생한다. 순환관계 데이터 모델의 예로는 조직, 사원, 메뉴 등이 있다.



사원	관리자
A	
B	A
C	A
D	C
E	C

(1) 순환관계 데이터 모델 (2) 계층형 구조 (3) 샘플 데이터

[그림 Ⅱ-2-6] 계층형 데이터

[그림 Ⅱ-2-6]은 사원에 대한 순환관계 데이터 모델을 표현한 것이다. (2)계층형 구조에서 A의 하위 사원은 B, C이고 B 밑에는 하위 사원이 없고 C의 하위 사원은 D, E가 있다. 계층형 구조를 데이터로 표현한 것이 (3)샘플 데이터이다. 계층형 데이터 조회는 DBMS 벤더와 버전에 따라 다른 방법으로 지원한다. 여기서는 Oracle과 SQL Server 기준으로 설명한다.

가. Oracle 계층형 질의

Oracle은 계층형 질의를 지원하기 위해서 [그림 Ⅱ-2-7]과 같은 계층형 질의 구문을 제공한다.

```
SELECT…
FROM 테이블
WHERE condition AND condition…
START WITH condition
CONNECT BY [NOCYCLE] condition AND condition…
[ORDER SIBLINGS BY column, column, …]
```

[그림 Ⅱ-2-7] 계층형 질의 구문

- START WITH절은 계층 구조 전개의 시작 위치를 지정하는 구문이다. 즉, 루트 데이터를 지정한다.(액세스) - CONNECT BY절은 다음에 전개될 자식 데이터를 지정하는 구문이다. 자식 데이터는 CONNECT BY절에 주어진 조건을 만족해야 한다.(조인) - PRIOR : CONNECT BY절에 사용되며, 현재 읽은 칼럼을 지정한다. PRIOR 자식 = 부모 형태를 사용하면 계층구조에서 자식 데이터에서 부모 데이터(자식 → 부모) 방향으로 전개하는 순방향 전개를 한다. 그리고 PRIOR 부모 = 자식 형태를 사용하면 반대로 부모 데이터에서 자식 데이터(부모 → 자식) 방향으로 전개하는 역방향 전개를 한다. - NOCYCLE : 데이터를 전개하면서 이미 나타났던 동일한 데이터가 전개 중에 다시 나타난다면 이것을 가리켜 사이클(Cycle)이 형성되었다라고 말한다. 사이클이 발생한 데이터는 런타임 오류가 발생한다. 그렇지만 NOCYCLE를 추가하면 사이클이 발생한 이후의 데이터는 전개하지 않는다. - ORDER SIBLINGS BY : 형제 노드(동일 LEVEL) 사이에서 정렬을 수행한다. - WHERE : 모든 전개를 수행한 후에 지정된 조건을 만족하는 데이터만 추출한다.(필터링)

Oracle은 계층형 질의를 사용할 때 다음과 같은 가상 칼럼(Pseudo Column)을 제공한다.

[표 Ⅱ-2-2] 계층형 질의에서 사용되는 가상 칼럼

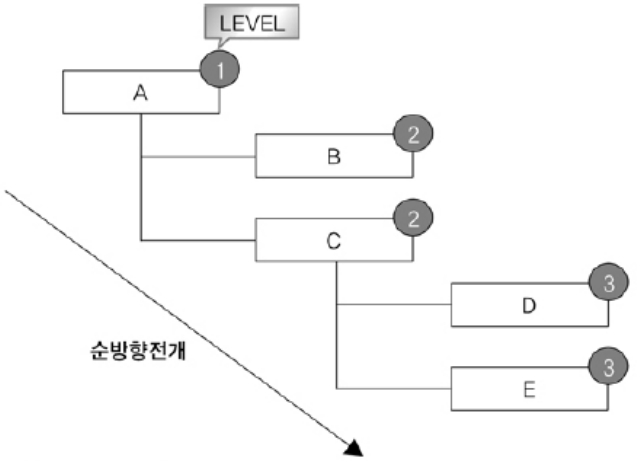
가상 칼럼	설명
LEVEL	루트 데이터이면 1, 그 하위 데이터이면 2이다. 리프(Leaf) 데이터까지 1씩 증가한다.
CONNECT_BY_ISLEAF	전개 과정에서 해당 데이터가 리프 데이터이면 1, 그렇지 않으면 0이다.
CONNECT_BY_ISCYCLE	전개 과정에서 자식을 갖는데, 해당 데이터가 조상으로서 존재하면 1, 그렇지 않으면 0이다. 여기서 조상이란 자신으로부터 루트까지의 경로에 존재하는 데이터를 말한다. CYCLE 옵션을 사용했을 때만 사용할 수 있다.

다음은 [그림 Ⅱ-2-6]의 (3)샘플 데이터를 계층형 질의 구문을 이용해서 조회한 것이다. 여기서는 결과 데이터를 들여쓰기 하기 위해서 LPAD 함수를 사용하였다.

[예제] SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1)) || 사원 사원, 관리자, CONNECT_BY_ISLEAF ISLEAF FROM 사원 START WITH 관리자 IS NULL CONNECT BY PRIOR 사원 = 관리자;

[실행 결과] LEVEL 사원 관리자 ISLEAF ----- -------- ----- ------ 1 A 0 2 B A 1 2 C A 0 3 D C 1 3 E C 1

A는 루트 데이터이기 때문에 레벨이 1이다. A의 하위 데이터인 B, C는 레벨이 2이다. 그리고 C의 하위 데이터인 D, E는 레벨이 3이다. 리프 데이터는 B, D, E이다. 관리자 → 사원 방향을 전개이기 때문에 순방향 전개이다. [그림 Ⅱ-2-8]은 계층형 질의에 대한 논리적인 실행 모습이다.



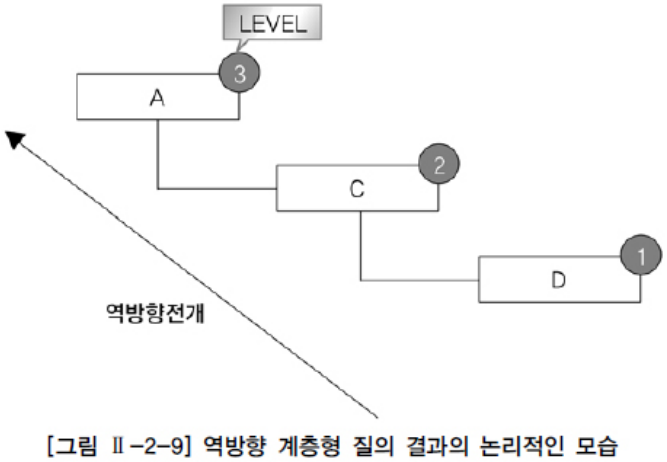
[그림 Ⅱ-2-8] 순방향 계층형 질의 결과의 논리적인 모습

다음 예제는 사원 'D'로부터 자신의 상위관리자를 찾는 역방향 전개의 예이다.

[예제] SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1)) || 사원 사원, 관리자, CONNECT_BY_ISLEAF ISLEAF FROM 사원 START WITH 사원 = 'D' CONNECT BY PRIOR 관리자 = 사원;

[실행 결과] LEVEL 사원 관리자 ISLEAF ----- --------- ----- ----- 1 D C 0 2 C A 0 3 A 1

본 예제는 역방향 전개이기 때문에 하위 데이터에서 상위 데이터로 전개된다. 결과를 보면 내용을 제외하고 표시 형태는 순방향 전개와 동일하다. D는 루트 데이터이기 때문에 레벨이 1이다. D의 상위 데이터인 C는 레벨이 2이다. 그리고 C의 상위 데이터인 A는 레벨이 3이다. 리프 데이터는 A이다. 루트 및 레벨은 전개되는 방향에 따라 반대가 됨을 알 수 있다. [그림 Ⅱ-2-9]는 역방향 전개에 대한 계층형 질의에 대한 논리적인 실행 모습이다.



[그림 Ⅱ-2-9] 역방향 계층형 질의 결과의 논리적인 모습

Orcle은 계층형 질의를 사용할 때 사용자 편의성을 제공하기 위해서 [표 Ⅱ-2-3]과 같은 함수를 제공한다

[표 Ⅱ-2-3] 계층형 질의에서 사용되는 함수

함수	설명
SYS_CONNECT_BY_PATH	루트 데이터부터 현재 전개할 데이터까지의 경로를 표시한다. 사용법 : SYS_CONNECT_BY_PATH(칼럼, 경로분리자)
CONNECT_BY_ROOT	현재 전개할 데이터의 루트 데이터를 표시한다. 단항 연산자이다. 사용법 : CONNECT_BY_ROOT 칼럼

SYS_CONNECT_BY_PATH, CONNECT_BY_ROOT를 사용한 예는 다음과 같다.

[예제] SELECT CONNECT_BY_ROOT 사원 루트사원, SYS_CONNECT_BY_PATH(사원, '/') 경로, 사원, 관리자 FROM 사원 START WITH 관리자 IS NULL CONNECT BY PRIOR 사원 = 관리자

[실행 결과] 루트사원 경로 사원 관리자 ------- ------- ---- ----- A /A A A /A/B B A A /A/C C A A /A/C/D D C A /A/C/E E C

START WITH를 통해 추출된 루트 데이터가 1건 이기 때문에 루트사원은 모두 A이다. 경로는 루트로부터 현재 데이터까지의 경로를 표시한다. 예를 들어, D의 경로는 A → C → D 이다.

나. SQL Server 계층형 질의

SQL Server 2000 버전까지는 계층형 질의를 작성할 수 있는 문법을 지원하지 않았다. 조직도처럼 계층적 구조를 가진 데이터는 저장 프로시저를 재귀 호출하거나 While 루프 문에서 임시 테이블을 사용하는 등 (순수한 쿼리가 아닌) 프로그램 방식으로 전개해야만 했다. 그러나 SQL Server 2005 버전부터는 하나의 질의로 원하는 결과를 얻을 수 있게 되었다. 먼저, Northwind 데이터베이스에 접속하여 Employees 테이블의 데이터를 조회해 보자.

USE NORTHWIND GO SELECT EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO FROM EMPLOYEES GO ********** EmployeeID LastName FirstName ReportsTo --------- -------- ------- -------- 1 Davolio Nancy 2 2 Fuller Andrew NULL 3 Leverling Janet 2 4 Peacock Margaret 2 5 Buchanan Steven 2 6 Suyama Michael 5 7 King Robert 5 8 Callahan Laura 2 9 Dodsworth Anne 5 (9개 행 적용됨)

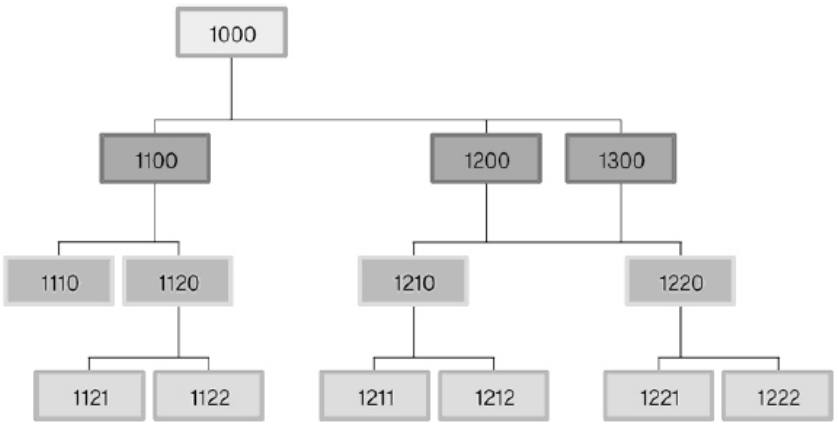
총 9개 로우가 있는데, ReportsTo 칼럼이 상위 사원에 해당하며 EmployeeID 칼럼과 재귀적 관계를 맺고 있다. EmployeeID가 2인 Fuller 사원을 살펴보면, ReportsTo 칼럼 값이 NULL이므로 계층 구조의 최상위에 있음을 알 수 있다. CTE(Common Table Expression)를 재귀 호출함으로써 Employees 데이터의 최상위부터 시작해 하위 방향으로 계층 구조를 전개하도록 작성한 쿼리와 결과는 다음과 같다.

WITH EMPLOYEES_ANCHOR AS (SELECT EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO, 0 AS LEVEL FROM EMPLOYEES WHERE REPORTSTO IS NULL /* 재귀 호출의 시작점 */ UNION ALL SELECT R.EMPLOYEEID, R.LASTNAME, R.FIRSTNAME, R.REPORTSTO, A.LEVEL + 1 FROM EMPLOYEES_ANCHOR A, EMPLOYEES R WHERE A.EMPLOYEEID = R.REPORTSTO) SELECT LEVEL, EMPLOYEEID, LASTNAME, FIRSTNAME, REPORTSTO FROM EMPLOYEES_ANCHOR GO ********** Level EmployeeID LastName FirstName ReportsTo ---- -------- ------- ----- -------- 0 2 Fuller Andrew NULL 1 1 Davolio Nancy 2 1 3 Leverling Janet 2 1 4 Peacock Margaret 2 1 5 Buchanan Steven 2 1 8 Callahan Laura 2 2 6 Suyama Michael 5 2 7 King Robert 5 2 9 Dodsworth Anne 5 (9개 행 적용됨)

WITH 절의 CTE 쿼리를 보면, UNION ALL 연산자로 쿼리 두 개를 결합했다. 둘 중 위에 있는 쿼리를 '앵커 멤버'(Anchor Member)라고 하고, 아래에 있는 쿼리를 '재귀 멤버'(Recursive Member)라고 한다. 아래는 재귀적 쿼리의 처리 과정이다.

1. CTE 식을 앵커 멤버와 재귀 멤버로 분할한다. 2. 앵커 멤버를 실행하여 첫 번째 호출 또는 기본 결과 집합(T0)을 만든다. 3. Ti는 입력으로 사용하고 Ti+1은 출력으로 사용하여 재귀 멤버를 실행한다. 4. 빈 집합이 반환될 때까지 3단계를 반복한다. 5. 결과 집합을 반환한다. 이것은 T0에서 Tn까지의 UNION ALL이다.

정리하자면 다음과 같다. 먼저, 앵커 멤버가 시작점이자 Outer 집합이 되어 Inner 집합인 재귀 멤버와 조인을 시작한다. 이어서, 앞서 조인한 결과가 다시 Outer 집합이 되어 재귀 멤버와 조인을 반복하다가 조인 결과가 비어 있으면 즉, 더 조인할 수 없으면 지금까지 만들어진 결과 집합을 모두 합하여 리턴한다. [그림 Ⅱ-2-10]에 있는 조직도를 쿼리로 출력했을 때, 대부분 사용자는 아래와 같은 결과를 기대할 것이다.(보기 편하도록 각 로우 앞쪽에 자신의 레벨만큼 빈칸을 삽입했다.)



[그림 Ⅱ-2-10] 조직도 예제

EmployeeID ManagerID -------- --------- 1000 NULL 1100 1000 1110 1100 1120 1100 1121 1120 1122 1120 1200 1000 1210 1200 1211 1210 1212 1210 1220 1200

```
1221 1220 1222 1220 1300 1000
```

아래에 t_emp 데이터의 최상위부터 시작해 하위 방향으로 계층 구조를 전개하도록 작성한 쿼리와 그 결과이다.

```
WITH T_EMP_ANCHOR AS ( SELECT EMPLOYEEID, MANAGERID, 0 AS LEVEL FROM T_EMP WHERE MANAGERID IS NULL /* 재귀 호출의 시작점 */ UNION ALL SELECT R.EMPLOYEEID, R.MANAGERID, A.LEVEL + 1 FROM T_EMP_ANCHOR A, T_EMP R WHERE A.EMPLOYEEID = R.MANAGERID ) SELECT LEVEL, REPLICATE(' ', LEVEL) + EMPLOYEEID AS EMPLOYEEID, MANAGERID FROM T_EMP_ANCHOR GO
********************************************************************** Level EmployeeID ManagerID --- ------- --------- 0 1000 NULL 1 1100 1000 1 1200 1000 1 1300 1000 2 1210 1200 2 1220 1200 3 1221 1220 3 1222 1220 3 1211 1210
```

```
3 1212 1210 2 1110 1100 2 1120 1100 3 1121 1120 3 1122 1120 (14개 행 적용됨
```

보다시피, 계층 구조를 단순히 하위 방향으로 전개했을 뿐 [그림 Ⅱ-2-10]에 있는 조직도와는 많이 다른 모습이다. 앞서 보았듯이, CTE 재귀 호출로 만들어낸 계층 구조는 실제와 다른 모습으로 출력된다. 따라서 조직도와 같은 모습으로 출력하려면 order by 절을 추가해 원하는 순서대로 결과를 정렬해야 한다. 실제 조직도와 같은 모습의 결과를 출력하도록, CTE에 Sort라는 정렬용 칼럼을 추가하고 쿼리 마지막에 order by 조건을 추가해보자.(단, 앵커 멤버와 재귀 멤버 양쪽에서 convert 함수 등으로 데이터 형식을 일치시켜야 한다.)

```
WITH T_EMP_ANCHOR AS ( SELECT EMPLOYEEID, MANAGERID, 0 AS LEVEL, CONVERT(VARCHAR(1000), EMPLOYEEID) AS SORT FROM T_EMP WHERE MANAGERID IS NULL /* 재귀 호출의 시작점 */ UNION ALL SELECT R.EMPLOYEEID, R.MANAGERID, A.LEVEL + 1, CONVERT(VARCHAR(1000), A.SORT + '/' + R.EMPLOYEEID) AS SORT FROM T_EMP_ANCHOR A, T_EMP R WHERE A.EMPLOYEEID = R.MANAGERID ) SELECT LEVEL, REPLICATE(' ', LEVEL) + EMPLOYEEID AS EMPLOYEEID, MANAGERID, SORT FROM T_EMP_ANCHOR ORDER BY SORT GO
```

CTE 안에서 Sort 칼럼에 사번(=EmployeeID)을 재귀적으로 더해 나가면 정렬 기준으로 삼을 수 있는 값이 만들어진다. 아래는 Sort 칼럼으로 정렬하여 출력한 결과인데, [그림 Ⅱ-2-10]에 있는 조직도의 모습과 일치한다.

```
Level EmployeeID ManagerID Sort ---- -------- -------- ------------ 0 1000 NULL 1000 1 1100 1000 1000/1100 2 1110 1100 1000/1100/1110 2 1120 1100 1000/1100/1120 3 1121 1120 1000/1100/1120/1121 3 1122 1120 1000/1100/1120/1122 1 1200 1000 1000/1200 2 1210 1200 1000/1200/1210 3 1211 1210 1000/1200/1210/1211 3 1212 1210 1000/1200/1210/1212 2 1220 1200 1000/1200/1220 3 1221 1220 1000/1200/1220/1221 3 1222 1220 1000/1200/1220/1222 1 1300 1000 1000/1300 (14개 행 적용됨)
```

가상의 Sort 칼럼을 추가해 정렬하는 게 아쉽기는 하지만, SQL Server에서 계층 구조를 실제 모습대로 출력하려면 현재(2005, 2008 버전 기준)로서는 감수해야 할 수밖에 없다.

2. 셀프 조인

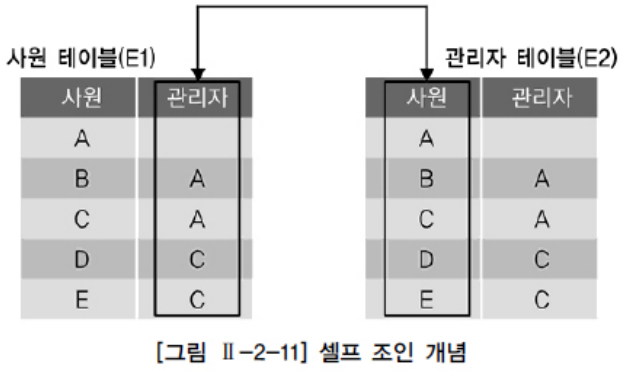
셀프 조인(Self Join)이란 동일 테이블 사이의 조인을 말한다. 따라서 FROM 절에 동일 테이블이 두 번 이상 나타난다. 동일 테이블 사이의 조인을 수행하면 테이블과 칼럼 이름이 모두 동일하기 때문에 식별을 위해 반드시 테이블 별칭(Alias)를 사용해야 한다. 그리고 칼럼에도 모두 테이블 별칭을 사용해서 어느 테이블의 칼럼인지 식별해줘야 한다. 이외 사항은 조인과 동일하다.

셀프 조인에 대한 기본적인 사용법은 다음과 같다

```
SELECT ALIAS명1.칼럼명, ALIAS명2.칼럼명, ... FROM 테이블1 ALIAS명1, 테이블2 ALIAS명2 WHERE ALIAS명1.칼럼명2 = ALIAS명2.칼럼명1;
```

```
SELECT WORKER.ID 사원번호, WORKER.NAME 사원명, MANAGER.NAME 관리자명 FROM EMP WORKER, EMP MANAGER WHERE WORKER.MGR = MANAGER.ID;
```

계층형 질의에서 살펴보았던 사원이라는 테이블 속에는 사원과 관리자가 모두 하나의 사원이라는 개념으로 동일시하여 같이 입력되어 있다. 이것을 이용해서 다음 문제를 셀프 조인으로 해결해 보면 다음과 같다. “자신과 상위, 차상위 관리자를 같은 줄에 표시하라.” 이 문제를 해결하기 위해서는 FROM 절에 사원 테이블을 두 번 사용해야 한다.



[그림 Ⅱ-2-11] 셀프 조인 개념

셀프 조인은 동일한 테이블(사원)이지만 [그림 Ⅱ-2-11]과 같이 개념적으로는 두 개의 서로 다른 테이블(사원, 관리자)을 사용하는 것과 동일하다. 동일 테이블을 다른 테이블인 것처럼 처리하기 위해 테이블 별칭을 사용한다. 여기서는 E1(사원), E2(관리자) 테이블 별칭을 사용하였다. 차상위 관리자를 구하기 위해서 E1.관리자 = E2.사원 조인 조건을 사용한다. 셀프 조인을 이용한 SQL문은 다음과 같다.

```
[예제] SELECT E1.사원, E1.관리자, E2.관리자 차상위_관리자 FROM 사원 E1, 사원 E2 WHERE E1.관리자 = E2.사원 ORDER BY E1.사원;
```

```
[실행 결과] 사원 관리자 차상위_관리자 ---- ------ ---------- B A C A D C A E C A
```

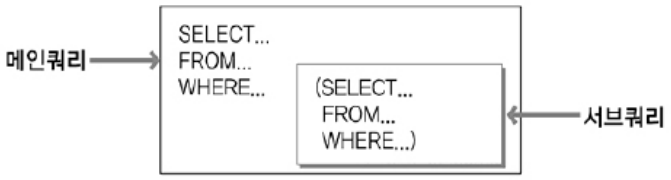
자신과 자신의 직속 관리자는 동일한 행에서 데이터를 구할 수 있으나 차상위 관리자는 바로 구할 수 없다. 차상위 관리자를 구하기 위해서는 자신의 직속 관리자를 기준으로 사원 테이블과 한번 더 조인(셀프 조인)을 수행해야 한다. 결과 표시를 위해 SELECT절에 2개의 '관리자' 칼럼을 사용되었다. 한 명은 자신의 직속 관리자(E1.관리자)이고 다른 한 명은 자신의 차상위 관리자(E2.관리자)이다. 결과를 보면, B와 C의 관리자는 A이고 차상위 관리자는 없다. D와 E의 관리자는 C이고 차상위 관리자는 A이다. 결과에서 A에 대한 정보는 누락되었다. 내부 조인(Inner Join)을 사용할 경우 자신의 관리자가 존재하지 않는 경우에는 관리자(E2) 테이블에서 조인할 대상이 존재하지 않기 때문에 해당 데이터는 결과에서 누락된다. 이를 방지하기 위해서는 아우터 조인을 사용해야 한다. 다음은 아우터 조인을 사용한 예이다.

[예제] SELECT E1.사원, E1.관리자, E2.관리자 차상위_관리자 FROM 사원 E1 LEFT OUTER JOIN 사원 E2 ON (E1.관리자 = E2.사원) ORDER BY E1.사원;

[실행 결과] 사원 관리자 차상위_관리자 ---- ----- ---------- A B A C A D C A E C A

아우터 조인을 사용해서 관리자가 존재하지 않는 데이터까지 모두 결과에 표시되었다.

서브쿼리(Subquery)란 하나의 SQL문안에 포함되어 있는 또 다른 SQL문을 말한다. 서브쿼리는 알려지지 않은 기준을 이용한 검색을 위해 사용한다. 서브쿼리는 [그림 Ⅱ-2-12]와 같이 메인쿼리가 서브쿼리를 포함하는 종속적인 관계이다.



[그림 Ⅱ-2-12] 메인쿼리와 서브쿼리

조인은 조인에 참여하는 모든 테이블이 대등한 관계에 있기 때문에 조인에 참여하는 모든 테이블의 칼럼을 어느 위치에서라도 자유롭게 사용할 수 있다. 그러나 서브쿼리는 메인쿼리의 칼럼을 모두 사용할 수 있지만 메인쿼리는 서브쿼리의 칼럼을 사용할 수 없다. 질의 결과에 서브쿼리 칼럼을 표시해야 한다면 조인 방식으로 변환하거나 함수, 스칼라 서브쿼리(Scalar Subquery) 등을 사용해야 한다. 조인은 집합간의 곱(Product)의 관계이다. 즉, 1:1 관계의 테이블이 조인하면 1(= 1 * 1) 레벨의 집합이 생성되고, 1:M 관계의 테이블을 조인하면 M(= 1 * M) 레벨의 집합이 생성된다. 그리고 M:N 관계의 테이블을 조인하면 MN(= M * N) 레벨의 집합이 결과로서 생성된다. 예를 들어, 조직(1)과 사원(M) 테이블을 조인하면 결과는 사원 레벨(M)의 집합이 생성된다. 그러나 서브쿼리는 서브쿼리 레벨과는 상관없이 항상 메인쿼리 레벨로 결과 집합이 생성된다. 예를 들어, 메인쿼리로 조직(1), 서브쿼리로 사원(M) 테이블을 사용하면 결과 집합은 조직(1) 레벨이 된다. SQL문에서 서브쿼리 방식을 사용해야 할 때 잘못 판단하여 조인 방식을 사용하는 경우가 있다. 예를 들어, 결과는 조직 레벨이고 사원 테이블에서 체크해야 할 조건이 존재한다고 가정하자. 이런 상황에서 SQL문을 작성할 때 조인을 사용한다면 결과 집합은 사원(M) 레벨이 될 것이다. 이렇게 되면 원하는 결과가 아니기 때문에 SQL문에 DISTINCT를 추가해서 결과를 다시 조직(1) 레벨로 만든다. 이와 같은 상황에서는 조인 방식이 아니라 서브쿼리 방식을 사용해야 한다. 메인쿼리로 조직을 사용하고 서브쿼리로 사원 테이블을 사용하면 결과 집합은 조직 레벨이 되기 때문에 원하는 결과가 된다.

서브쿼리를 사용할 때 다음 사항에 주의해야 한다.

① 서브쿼리를 괄호로 감싸서 사용한다. ② 서브쿼리는 단일 행(Single Row) 또는 복수 행(Multiple Row) 비교 연산자와 함께 사용 가능하다. 단일 행 비교 연산자는 서브쿼리의 결과가 반드시 1건 이하이어야 하고 복수 행 비교 연산자는 서브쿼리의 결과 건수와 상관 없다. ③ 서브쿼리에서는 ORDER BY를 사용하지 못한다. ORDER BY절은 SELECT절에서 오직 한 개만 올 수 있기 때문에 ORDER BY절은 메인쿼리의 마지막 문장에 위치해야 한다.

서브쿼리가 SQL문에서 사용이 가능한 곳은 다음과 같다.

- SELECT 절 - FROM 절 - WHERE 절 - HAVING 절 - ORDER BY 절 - INSERT문의 VALUES 절 - UPDATE문의 SET 절

서브쿼리의 종류는 동작하는 방식이나 반환되는 데이터의 형태에 따라 분류할 수 있다. 동작하는 방식에 따라 서브쿼리를 분류하면 [표 Ⅱ-2-4]와 같이 두 가지로 나눌 수 있다.

[표 Ⅱ-2-4] 동작하는 방식에 따른 서브쿼리 분류

서브쿼리 종류	설명
Un-Correlated(비연관) 서브쿼리	서브쿼리가 메인쿼리 칼럼을 가지고 있지 않는 형태의 서브쿼리이다. 메인쿼리에 값(서브쿼리가 실행된 결과)을 제공하기 위한 목적으로 주로 사용한다.
Correlated(연관) 서브쿼리	서브쿼리가 메인쿼리 칼럼을 가지고 있는 형태의 서브쿼리이다. 일반적으로 메인쿼리가 먼저 수행되어 읽혀진 데이터를 서브쿼리에서 조건이 맞는지 확인하고자 할 때 주로 사용된다.

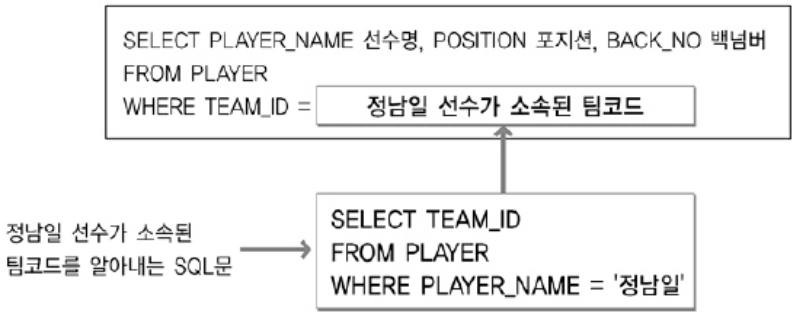
서브쿼리는 메인쿼리 안에 포함된 종속적인 관계이기 때문에 논리적인 실행순서는 항상 메인쿼리에서 읽혀진 데이터에 대해 서브쿼리에서 해당 조건이 만족하지를 확인하는 방식으로 수행되어야 한다. 그러나 실제 서브쿼리의 실행순서는 상황에 따라 달라질 수 있다. 반환되는 데이터의 형태에 따라 서브쿼리는 [표 Ⅱ-2-5]와 같이 세가지로 분류된다.

[표 Ⅱ-2-5] 반환되는 데이터의 형태에 따른 서브쿼리 분류

서브쿼리 종류	설명
Single Row 서브쿼리 (단일 행 서브쿼리)	서브쿼리의 실행 결과가 항상 1건 이하인 서브쿼리를 의미한다. 단일 행 서브쿼리는 단일 행 비교 연산자와 함께 사용된다. 단일 행 비교 연산자에는 =, 〈, 〈=, 〉, 〉=, ◇이 있다.
Multi Row 서브쿼리 (다중 행 서브쿼리)	서브쿼리의 실행 결과가 여러 건인 서브쿼리를 의미한다. 다중 행 서브쿼리는 다중 행 비교 연산자와 함께 사용된다. 다중 행 비교 연산자에는 IN, ALL, ANY, SOME, EXISTS가 있다.
Multi Column 서브쿼리 (다중 칼럼 서브쿼리)	서브쿼리의 실행 결과로 여러 칼럼을 반환한다. 메인쿼리의 조건절에 여러 칼럼을 동시에 비교할 수 있다. 서브쿼리와 메인쿼리에서 비교하고자 하는 칼럼 개수와 칼럼의 위치가 동일해야 한다.

1. 단일 행 서브 쿼리

서브쿼리가 단일 행 비교 연산자(=, <, <=, >, >=, <>)와 함께 사용할 때는 서브쿼리의 결과 건수가 반드시 1건 이하이어야 한다. 만약, 서브쿼리의 결과 건수가 2건 이상을 반환하면 SQL문은 실행시간(Run Time) 오류가 발생한다. 이런 종류의 오류는 컴파일 할 때(Compile Time)는 알 수 없는 오류이다. 단일 행 서브쿼리의 예로 '정남일' 선수가 소속된 팀의 선수들에 대한 정보를 표시하는 문제를 가지고 설명해 보면 다음과 같다.



[그림 Ⅱ-2-13] 단일 행 서브쿼리의 예제1

[그림 Ⅱ-2-13]은 2개의 SQL문으로 구성되어 있다. 정남일 선수의 소속팀을 알아내는 SQL문(서브쿼리 부분)과 이 결과를 이용해서 해당 팀에 소속된 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)으로 구성된다. 정남일 선수가 소속된 팀의 선수들에 대한 정보를 표시하는 문제를 서브쿼리 방식의 SQL문으로 작성하면 다음과 같다.

```
[예제] SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버 FROM PLAYER WHERE TEAM_ID = (SELECT TEAM_ID FROM PLAYER WHERE PLAYER_NAME = '정남일') ORDER BY PLAYER_NAME;
```

```
[실행 결과] 선수명 포지션 백넘버 ------- ----- ----- 강철 DF 3 김반 MF 14 김영수 MF 30 김정래 GK 33 김창원 DF 5 김회택 TM 꼬레아 FW 16 노병준 MF 22 51개의 행이 선택되었다.
```

정남일 선수의 소속팀을 알아내는 서브쿼리가 먼저 수행되어 정남일 선수의 소속팀 코드가 반환된다. 메인쿼리는서브쿼리에서 반환된 결과를 이용해서 조건을 만족하는 선수들의 정보를 출력한다. 만약, 정남일 선수가 동명이인이었다면 2건 이상의 결과가 반환되어 SQL문은 오류가 발생될 것이다. 테이블 전체에 하나의 그룹함수를 적용할 때는 그 결과값이 1건이 생성되기 때문에 단일 행 서브쿼리로서 사용 가능하다. 선수들 중에서 키가 평균 이하인 선수들의 정보를 출력하는 문제를 가지고 그룹함수를 사용한 서브쿼리를 알아보도록 한다.

[그림 Ⅱ-2-14] 단일 행 서브쿼리의 예제2

[그림 Ⅱ-2-14]는 2개의 SQL문으로 구성되어 있다. 선수들의 평균키를 알아내는 SQL문(서브쿼리 부분)과 이 결과를 이용해서 키가 평균 이하의 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)으로 구성된다. [그림 Ⅱ-2-14]를 SQL문으로 작성하면 다음과 같다.

```
[예제] SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버 FROM PLAYER WHERE HEIGHT <= (SELECT AVG(HEIGHT) FROM PLAYER) ORDER BY PLAYER_NAME;
```

```
[실행 결과] 선수명 포지션 백넘버 ------- ------ ----- 가비 MF 10 강대희 MF 26 강용 DF 2 강정훈 MF 38 강철 DF 3 고규억 DF 29 고민기 FW 24 고종수 MF 22 228개의 행이 선택되었다.
```

12. 다중 행 서브쿼리

서브쿼리의 결과가 2건 이상 반환될 수 있다면 반드시 다중 행 비교 연산자(IN, ALL, ANY, SOME)와 함께 사용해야 한다. 그렇지 않으면 SQL문은 오류를 반환한다. 다중 행 비교 연산자는 다음과 같다.

[표 Ⅱ-2-6] 다중 행 비교 연산자

다중 행 연산자	설명
IN (서브쿼리)	서브쿼리의 결과에 존재하는 임의의 값과 동일한 조건을 의미한다. (Multiple OR 조건)
비교연산자 ALL (서브쿼리)	서브쿼리의 결과에 존재하는 모든 값을 만족하는 조건을 의미한다. 비교 연산자로 ">"를 사용했다면 메인쿼리는 서브쿼리의 모든 결과 값을 만족해야 하므로, 서브쿼리 결과의 최대값보다 큰 모든 건이 조건을 만족한다.
비교연산자 ANY (서브쿼리)	서브쿼리의 결과에 존재하는 어느 하나의 값이라도 만족하는 조건을 의미한다. 비교 연산자로 ">"를 사용했다면 메인쿼리는 서브쿼리의 값들 중 어떤 값이라도 만족하면 되므로, 서브쿼리의 결과의 최소값보다 큰 모든 건이 조건을 만족한다. (SOME은 ANY와 동일함)
EXISTS (서브쿼리)	서브쿼리의 결과를 만족하는 값이 존재하는지 여부를 확인하는조건을 의미한다. 조건을 [출처] 서브쿼리의 종류|작성자 찐 만족하는 건이 여러 건이더라도 1건만 찾으면 더 이상 검색하지 않는다.

선수들 중에서 '정현수'라는 선수가 소속되어 있는 팀 정보를 출력하는 서브쿼리를 작성하면 다음과 같다.

[예제] SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명 FROM TEAM WHERE TEAM_ID = (SELECT TEAM_ID FROM PLAYER WHERE PLAYER_NAME = '정현수') ORDER BY TEAM_NAME; ORA-01427: 단일 행 하위 질의에 2개 이상의 행이 리턴되었다.

위의 SQL문은 서브쿼리의 결과로 2개 이상의 행이 반환되어 단일 행 비교 연산자인 '='로는 처리가 불가능하기 때문에 에러가 반환되었다. 따라서 다중 행 비교 연산자로 바꾸어서 SQL문을 작성하면 다음과 같다.

[예제] SELECT REGION_NAME 연고지명, TEAM_NAME 팀명, E_TEAM_NAME 영문팀명 FROM TEAM WHERE TEAM_ID IN (SELECT TEAM_ID FROM PLAYER WHERE PLAYER_NAME = '정현수') ORDER BY TEAM_NAME;

[실행 결과] 연고지명 팀명 영문팀명 ------ ----- ---------- 전남 드래곤즈 CHUNNAM DRAGONS FC 성남 일화천마 SEONGNAM ILHWA CHUNMA FC 2개의 행이 선택되었다.

실행 결과를 보면 '정현수'란 이름을 가진 선수가 두 명이 존재한다. 소속팀은 각각 전남 드래곤즈팀(K07)과 성남 일화천마팀(K08)이다. 본 예제에서는 동명이인에 대한 내용을 예로 들었지만, 서브쿼리의 실행 결과가 2건 이상이 나오는 모든 경우에 다중 행 비교 연산자를 사용해야 한다.

3. 다중 칼럼 서브쿼리

다중 칼럼 서브쿼리는 서브쿼리의 결과로 여러 개의 칼럼이 반환되어 메인쿼리의 조건과 동시에 비교되는 것을 의미한다. 소속팀별 키가 가장 작은 사람들의 정보를 출력하는 문제를 가지고 다중 칼럼 서브쿼리를 알아보도록 한다. 소속팀별 키가 가장 작은 사람들의 정보는 GROUP BY를 이용하여 찾을 수 있으므로 다음과 같이 SQL문을 작성할 수 있다.

[예제] SELECT TEAM_ID 팀코드, PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM PLAYER WHERE (TEAM_ID, HEIGHT) IN (SELECT TEAM_ID, MIN(HEIGHT) FROM PLAYER GROUP BY TEAM_ID) ORDER BY TEAM_ID, PLAYER_NAME;

[실행 결과] 팀코드 선수명 포지션 백넘버 키 ----- -------- ------ ---- --- K01 마르코스 FW 44 170 K01 박정수 MF 8 170 K02 고창현 MF 8 170 K02 정준 MF 44 170 K03 김중규 MF 42 170 19개의 행이 선택되었다.

SQL문의 실행 결과를 보면 서브쿼리의 결과값으로 소속팀코드(TEAM_ID)와 소속팀별 가장 작은 키를 의미하는 MIN(HEIGHT)라는 두 개의 칼럼을 반환했다. 메인쿼리에서는 조건절에 TEAM_ID와 HEIGHT 칼럼을 괄호로 묶어서 서브쿼리 결과와 비교하여 원하는 결과를 얻었다. 실행 결과에서 보면 하나 팀에서 키가 제일 작은 선수 한 명씩만 반환된 것이 아니라 같은 팀에서 여러 명이 반환된 것을 확인할 수 있다. 이것은 동일 팀 내에서 조건(팀별 가장 작은 키)을 만족하는 선수가 여러 명이 존재하기 때문이다. 그러나 이 기능은 SQL Server에서는 지원되지 않는 기능이다.

4. 연관 서브쿼리

연관 서브쿼리(Correlated Subquery)는 서브쿼리 내에 메인쿼리 칼럼이 사용된 서브쿼리이다. 선수 자신이 속한 팀의 평균 키보다 작은 선수들의 정보를 출력하는 SQL문을 연관 서브쿼리를 이용해서 작성해 보면 다음과 같다

[예제] SELECT T.TEAM_NAME 팀명, M.PLAYER_NAME 선수명, M.POSITION 포지션, M.BACK_NO 백넘버, M.HEIGHT 키 FROM PLAYER M, TEAM T WHERE M.TEAM_ID = T.TEAM_ID AND M.HEIGHT < (SELECT AVG(S.HEIGHT) FROM PLAYER S WHERE S.TEAM_ID = M.TEAM_ID AND S.HEIGHT IS NOT NULL GROUP BY S.TEAM_ID) ORDER BY 선수명;

[실행 결과] 팀명 선수명 포지션 백넘버 키 -------- ----- ----- ----- -- 삼성블루윙즈 가비 MF 10 177 삼성블루윙즈 강대희 MF 26 174 스틸러스 강용 DF 2 179 시티즌 강정훈 MF 38 175 드래곤즈 강철 DF 3 178 현대모터스 고관영 MF 32 180 현대모터스 고민기 FW 24 178 삼성블루윙즈 고종수 MF 22 176 224의 행이 선택되었다.

예를 들어, 가비 선수는 삼성블루윙즈팀 소속이므로 삼성블루윙즈팀 소속의 평균키를 구하고 그 평균키와 가비 선수의 키를 비교하여 적을 경우에 선수에 대한 정보를 출력한다. 만약, 평균키 보다 선수의 키가 크거나 같으면 조건에 맞지 않기 때문에 해당 데이터는 출력되지 않는다. 이와 같은 작업을 메인쿼리에 존재하는 모든 행에 대해서 반복 수행한다. EXISTS 서브쿼리는 항상 연관 서브쿼리로 사용된다. 또한 EXISTS 서브쿼리의 특징은 아무리 조건을 만족하는

건이 여러 건이더라도 조건을 만족하는 1건만 찾으면 추가적인 검색을 진행하지 않는다. 다음은 EXISTS 서브쿼리를 사용하여 '20120501' 부터 '20120502' 사이에 경기가 있는 경기장을 조회하는 SQL문이다.

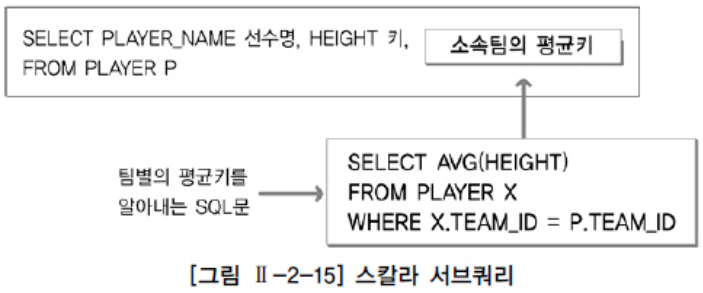
[예제] SELECT STADIUM_ID ID, STADIUM_NAME 경기장명 FROM STADIUM A WHERE EXISTS (SELECT 1 FROM SCHEDULE X WHERE X.STADIUM_ID = A.STADIUM_ID AND X.SCHE_DATE BETWEEN '20120501' AND '20120502')

[실행 결과] ID 경기장명 --- ---------- B01 인천월드컵경기장 B04 수원월드컵경기장 B05 서울월드컵경기장 C02 부산아시아드경기장 4개의 행이 선택되었다.

5. 그밖에 위치에서 사용하는 서브쿼리

가. SELECT 절에 서브쿼리 사용하기

다음은 SELECT 절에서 사용하는 서브쿼리인 스칼라 서브쿼리(Scalar Subquery)에 대해서 알아본다. 스칼라 서브쿼리는 한 행, 한 칼럼(1 Row 1 Column)만을 반환하는 서브쿼리를 말한다. 스칼라 서브쿼리는 칼럼을 쓸 수 있는 대부분의 곳에서 사용할 수 있다. 선수 정보와 해당 선수가 속한 팀의 평균 키를 함께 출력하는 예제로 스칼라 서브쿼리를 설명하면 다음과 같다.



[그림 Ⅱ-2-15] 스칼라 서브쿼리

[그림 Ⅱ-2-15]는 2개의 SQL문으로 구성되어 있다. 선수들의 정보를 출력하는 SQL문(메인쿼리 부분)과 해당 선수의 소속팀별 평균키를 알아내는 SQL문(서브쿼리 부분)으로 구성된다. 여기서 선수의 소속팀별 평균키를 알아내는 스칼라 서브쿼리는 메인쿼리의 결과 건수만큼 반복수행 된다. [그림 Ⅱ-2-15]를 SQL문으로 작성하면 다음과 같다.

[예제] SELECT PLAYER_NAME 선수명, HEIGHT 키, (SELECT AVG(HEIGHT) FROM PLAYER X WHERE X.TEAM_ID = P.TEAM_ID) 팀평균키 FROM PLAYER P

[실행 결과] 선수명 키 팀평균키 ------- ---- ---------- 가비 177 179.067 가이모토 182 178.854 강대희 174 179.067 강성일 182 177.485 강용 179 179.911 강정훈 175 177.485 강철 178 178.391 고관영 180 180.422 480개의 행이 선택되었다.

스칼라 서브쿼리 또한 단일 행 서브쿼리이기 때문에 결과가 2건 이상 반환되면 SQL문은 오류를 반환한다.

나. FROM 절에서 서브쿼리 사용하기

FROM 절에서 사용되는 서브쿼리를 인라인 뷰(Inline View)라고 한다. FROM 절에는 테이블 명이 오도록 되어 있다. 그런데 서브쿼리가 FROM 절에 사용되면 어떻게 될까? 서브쿼리의 결과가 마치 실행 시에 동적으로 생성된 테이블인 것처럼 사용할 수 있다. 인라인 뷰는 SQL문이 실행될 때만 임시적으로 생성되는 동적인 뷰이기 때문에 데이터베이스에 해당 정보가 저장되지 않는다. 그래서 일반적인 뷰를 정적 뷰(Static View)라고 하고 인라인 뷰를 동적 뷰(Dynamic View)라고도 한다. 뷰에 대해서는 뒤에서 좀더 설명하기로 한다. 인라인 뷰는 테이블 명이 올 수 있는 곳에서 사용할 수 있다. 서브쿼리의 칼럼은 메인쿼리에서 사용할 수 없다고 했다. 그러나 인라인 뷰는 동적으로 생성된 테이블이다. 인라인 뷰를 사용하는 것은 조인 방식을 사용하는 것과 같다. 그렇기 때문에 인라인 뷰의 칼럼은 SQL문 자유롭게 참조할 수 있다. K-리그 선수들 중에서 포지션이 미드필더(MF)인 선수들의 소속팀명 및 선수 정보를 출력하고자 한다. 인라인 뷰를 활용해서 SQL문을 만들어 보자.

[예제] SELECT T.TEAM_NAME 팀명, P.PLAYER_NAME 선수명, P.BACK_NO 백넘버 FROM (SELECT TEAM_ID, PLAYER_NAME, BACK_NO FROM PLAYER WHERE POSITION = 'MF') P, TEAM T WHERE P.TEAM_ID = T.TEAM_ID ORDER BY 선수명;

[실행 결과] 팀명 선수명 백넘버 --------- ------- ----- 삼성블루윙즈 가비 10 삼성블루윙즈 강대희 26 시티즌 강정훈 38 현대모터스 고관영 32 삼성블루윙즈 고종수 22 삼성블루윙즈 고창현 8 시티즌 공오균 22 일화천마 곽치국 32 162개의 행이 선택되었다.

SQL문을 보면 선수들 중에서 포지션이 미드필더(MF) 선수들을 인라인 뷰를 통해서 추출하고 인라인 뷰의 결과와 TEAM 테이블과 조인해서 팀명(TEAM_NAME)을 출력하고 있다. 인라인 뷰에서는 ORDER BY절을 사용할 수 있다. 인라인 뷰에 먼저 정렬을 수행하고 정렬된 결과 중에서 일부 데이터를 추출하는 것을 TOP-N 쿼리라고 한다. TOP-N 쿼리를 수행하기 위해서는 정렬 작업과 정렬 결과 중에서 일부 데이터만을 추출할 수 있는 방법이 필요하다. Oracle에서는 ROWNUM이라는 연산자를 통해서 결과로 추출하고자 하는 데이터 건수를 제약할 수 있다.

[예제] Oracle SELECT PLAYER_NAME 선수명, POSITION 포지션, BACK_NO 백넘버, HEIGHT 키 FROM (SELECT PLAYER_NAME, POSITION, BACK_NO, HEIGHT FROM PLAYER WHERE HEIGHT IS NOT NULL ORDER BY HEIGHT DESC) WHERE ROWNUM <= 5;

[예제] SQL Server SELECT TOP(5) PLAYER_NAME AS 선수명, POSITION AS 포지션, BACK_NO AS 백넘버, HEIGHT AS 키 FROM PLAYER WHERE HEIGHT IS NOT NULL ORDER BY HEIGHT DESC

[실행 결과] 선수명 포지션 백넘버 키 -------- ----- --- --- 서동명 GK 21 196 권정혁 GK 1 195 김석 FW 20 194 정경두 GK 41 194 이현 GK 1 192 5개의 행이 선택되었다.

당 SQL문의 인라인 뷰에서 선수의 키를 내림차순으로 정렬(가장 키가 큰 선수부터 출력)한 후 메인쿼리에서 ROWNUM을 사용해서 5명의 선수의 정보만을 추출하였다. 이것은 모든 선수들 중에서 가장 키가 큰 5명의 선수를 출력한 것이다. 만약, 다른 선수 중에서 키가 192인 선수가 더 존재하더라도 해당 SQL문

에서는 데이터가 출력되지 않는다. 이런 데이터까지 추출하고자 한다면 분석함수의 RANK관련 함수를 사용해야 한다.

다. HAVING 절에서 서브쿼리 사용하기

HAVING 절은 그룹함수와 함께 사용될 때 그룹핑된 결과에 대해 부가적인 조건을 주기 위해서 사용한다. 평균키가 삼성 블루윙즈팀의 평균키보다 작은 팀의 이름과 해당 팀의 평균키를 구하는 SQL문을 작성하면 다음과 같다.

```
[예제] SELECT P.TEAM_ID 팀코드, T.TEAM_NAME 팀명, AVG(P.HEIGHT) 평균키 FROM PLAYER P, TEAM T WHERE P.TEAM_ID = T.TEAM_ID GROUP BY P.TEA
M_ID, T.TEAM_NAME HAVING AVG(P.HEIGHT) < (SELECT AVG(HEIGHT) FROM PLAYER WHERE TEAM_ID ='K02')
```

```
[실행 결과] 팀코드 팀명 평균키 ---- ---------- ------ K13 강원FC 173.667 K15 대구FC 175.333 K11 경남FC 176.333 K14 제주유나이티드FC 169.5 K12 광주상무 173.5 K0
7 드래곤즈 178.391 K08 일화천마 178.854 K10 시티즌 177.485 8개의 행이 선택되었다.
```

라. UPDATE문의 SET 절에서 사용하기

현재 TEAM 테이블에는 STADIUM_NAME 칼럼이 없다. TEAM 테이블에 STADIUM_NAME을 추가(ALTER TABLE ADD COLUMN)하였다고 가정하자. TEA
M 테이블에 추가된 STADIUM_NAME의 값을 STADIUM 테이블을 이용하여 변경하고자 할 때 다음과 같이 SQL문을 작성할 수 있다.

```
UPDATE TEAM A SET A.STADIUM_NAME = (SELECT X.STADIUM_NAME FROM STADIUM X WHERE X.STADIUM_ID = A.STADIUM_ID);
```

서브쿼리를 사용한 변경 작업을 할 때 서브쿼리의 결과가 NULL을 반환할 경우 해당 컬럼의 결과가 NULL이 될 수 있기 때문에 주의해야 한다.

마. INSERT문의 VALUES절에서 사용하기

PLAYER 테이블에 '홍길동'이라는 선수를 삽입하고자 한다. 이때 PLAYER_ID의 값을 현재 사용중인 PLAYER_ID에 1을 더한 값으로 넣고자 한다. 다음과 같이 SQL문을 SQL문을 작성할 수 있다.

```
INSERT INTO PLAYER(PLAYER_ID, PLAYER_NAME, TEAM_ID) VALUES((SELECT TO_CHAR(MAX(TO_NUMBER(PLAYER_ID))+1) FROM PLAYER), '홍길동', 'K0
6');
```

6. 뷰(View)

테이블은 실제로 데이터를 가지고 있는 반면, 뷰(View)는 실제 데이터를 가지고 있지 않다. 뷰는 단지 뷰 정의(View Definition)만을 가지고 있다. 질의에서 뷰가 사용되면 뷰 정의를 참조해서 DBMS 내부적으로 질의를 재작성(Rewrite)하여 질의를 수행한다. 뷰는 실제 데이터를 가지고 있지 않지만 테이블이 수행하는 역할을 수행하기 때문에 가상 테이블(Virtual Table)이라고도 한다. 뷰는 [표 II-2-7]과 같은 장점을 갖는다.

[표 Ⅱ-2-7] 뷰 사용의 장점

뷰의 장점	설명
독립성	테이블 구조가 변경되어도 뷰를 사용하는 응용 프로그램은 변경하지 않아도 된다.
편리성	복잡한 질의를 뷰로 생성함으로써 관련 질의를 단순하게 작성할 수 있다. 또한 해당 형태의 SQL문을 자주 사용할 때 뷰를 이용하면 편리하게 사용할 수 있다.
보안성	직원의 급여정보와 같이 숨기고 싶은 정보가 존재한다면, 뷰를 생성할 때 해당 칼럼을 빼고 생성함으로써 사용자에게 정보를 감출 수 있다.

뷰는 다음과 같이 CREATE VIEW문을 통해서 생성할 수 있다.

```
CREATE VIEW V_PLAYER_TEAM AS SELECT P.PLAYER_NAME, P.POSITION, P.BACK_NO, P.TEAM_ID, T.TEAM_NAME FROM PLAYER P, TEAM T WHERE P.T
EAM_ID = T.TEAM_ID;
```

해당 뷰는 선수 정보와 해당 선수가 속한 팀명을 함께 추출하는 것이다. 뷰의 명칭은 'V_PLAYER_TEAM'이다. 뷰는 테이블뿐만 아니라 이미 존재하는 뷰를 참조해서도 생성할 수 있다.

```
CREATE VIEW V_PLAYER_TEAM_FILTER AS SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_NAME FROM V_PLAYER_TEAM WHERE POSITION IN ('G
K', 'MF');
```

V_PLAYER_TEAM_FILTER 뷰는 이미 앞에서 생성했던 V_PLAYER_TEAM 뷰를 기반으로 해서 생성된 뷰다. V_PLAYER_TEAM_FILTER 뷰는 선수 포지션이 골키퍼(GK), 미드필더(MF)인 선수만을 추출하고자 하는 뷰이다.(뷰를 포함하는 뷰를 잘못 생성하는 경우 성능상의 문제를 유발할 수 있으므로, 뷰와 SQL의 수행원리를 잘 이해하고 사용하기 바란다) 뷰를 사용하기 위해서는 해당 뷰의 이름을 이용하면 된다. 뷰를 사용하는 방법은 다음과 같다.

```
[예제] SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_ID, TEAM_NAME FROM V_PLAYER_TEAM WHERE PLAYER_NAME LIKE '황%'
```

```
[실행 결과] PLAYER_NAME POSITION BACK_NO TEAM ID TEAM_NAME ----------- ------- ------- ------- --------- 황철민 MF 35 K06 아이파크 황승주 DF 98 K05 현대모터스
황연석 FW 16 K08 일화천마 3개의 행이 선택되었다.
```

이것은 V_PLAYER_TEAM 뷰에서 성이 '황'씨인 선수만을 추출하는 SQL문이다. 결과로서 3건이 추출되었다. 뷰를 사용하는 경우에는 DBMS가 내부적으로 SQL문을 다음과 같이 재작성한다.

```
SELECT PLAYER_NAME, POSITION, BACK_NO, TEAM_ID, TEAM_NAME FROM (SELECT P.PLAYER_NAME, P.POSITION, P.BACK_NO, P.TEAM_ID, T.TEAM_NA
ME FROM PLAYER P, TEAM T WHERE P.TEAM_ID = T.TEAM_ID) WHERE PLAYER_NAME LIKE '황%'
```

이것은 앞에서 설명했던 인라인 뷰와 유사한 모습임을 알 수 있다. 이와 같은 형태로 사용되기 때문에 뷰는 데이터를 저장하지 않고도 데이터를 조회할 수 있

다. 뷰를 제거하기 위해서는 DROP VIEW문을 사용한다.

```
DROP VIEW V_PLAYER_TEAM; DROP VIEW V_PLAYER_TEAM_FILTER;
```

1. 데이터 분석 개요

ANSI/ISO SQL 표준은 데이터 분석을 위해서 다음 세 가지 함수를 정의하고 있다.

- AGGREGATE FUNCTION - GROUP FUNCTION - WINDOW FUNCTION

- AGGREGATE FUNCTION

GROUP AGGREGATE FUNCTION이라고도 부르며, GROUP FUNCTION의 한 부분으로 분류할 수 있다. 1장 7절에서 설명한 COUNT, SUM, AVG, MAX, MIN 외 각종 집계 함수들이 포함되어 있다.

- GROUP FUNCTION

결산 개념의 업무를 가지는 원가나 판매 시스템의 경우는 소계, 중계, 합계, 총 합계 등 여러 레벨의 결산 보고서를 만드는 것이 중요 업무 중의 하나이다. 개발자들이 이런 보고서를 작성하기 위해서는 SQL이 포함된 3GL으로 배치 프로그램을 작성하거나, 레벨별 집계를 위한 여러 단계의 SQL을 UNION, UNION ALL로 묶은 후 하나의 테이블을 여러 번 읽어 다시 재정렬하는 복잡한 단계를 거쳐야 했다. 그러나 그룹 함수를 사용한다면 하나의 SQL로 테이블을 한 번만 읽어서 빠르게 원하는 리포트를 작성할 수 있다. 추가로, 소계/합계를 표시하기 위해 GROUPING 함수와 CASE 함수를 이용하면 쉽게 원하는 포맷의 보고서 작성도 가능하다. 그룹 함수로는 집계 함수를 제외하고, 소그룹 간의 소계를 계산하는 ROLLUP 함수, GROUP BY 항목들 간 다차원적인 소계를 계산 할 수 있는 CUBE 함수, 특정 항목에 대한 소계를 계산하는 GROUPING SETS 함수가 있다. ROLLUP은 GROUP BY의 확장된 형태로 사용하기가 쉬우며 병렬로 수행이 가능하기 때문에 매우 효과적일 뿐 아니라 시간 및 지역처럼 계층적 분류를 포함하고 있는 데이터의 집계에 적합하도록 되어 있다. CUBE는 결합 가능한 모든 값에 대하여 다차원적인 집계를 생성하게 되므로 ROLLUP에 비해 다양한 데이터를 얻는 장점이 있는 반면에, 시스템에 부하를 많이 주는 단점이 있다. GROUPING SETS는 원하는 부분의 소계만 손쉽게 추출할 수 있는 장점이 있다. ROLLUP, CUBE, GROUPING SETS 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 정렬 칼럼을 명시해야 한다.

- WINDOW FUNCTION

분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수는 데이터웨어하우스에서 발전한 기능이며, 자세한 내용은 다음 절에서 설명한다.

2. ROLLUP 함수

ROLLUP에 지정된 Grouping Columns의 List는 Subtotal을 생성하기 위해 사용되어지며, Grouping Columns의 수를 N이라고 했을 때 N+1 Level의 Subtotal이 생성된다. 중요한 것은, ROLLUP의 인수는 계층 구조이므로 인수 순서가 바뀌면 수행 결과도 바뀌게 되므로 인수의 순서에도 주의해야 한다. ROLLUP과 CUBE의 효과를 알아보기 위해 단계별로 데이터를 출력해본다.

STEP 1. 일반적인 GROUP BY 절 사용

[예제] 부서명과 업무명을 기준으로 사원수와 급여 합을 집계한 일반적인 GROUP BY SQL 문장을 수행한다.

```
[예제] SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME, JOB;
```

```
[실행 결과] DNAME JOB Total Empl Total Sal --------- -------- ------- ------- SALES MANAGER 1 2850 SALES CLERK 1 950 ACCOUNTING MANAGER 1 2450 RESEARCH ANALYST 2 6000 ACCOUNTING CLERK 1 1300 SALES SALESMAN 4 5600 RESEARCH MANAGER 1 2975 ACCOUNTING PRESIDENT 1 5000 RESEARCH CLERK 2 1900 9개의 행이 선택되었다.
```

Oracle을 포함한 일부 DBMS의 과거 버전에서는 GROUP BY 절 사용시 자동적으로 정렬을 수행하였으나, 현재 대부분의 DBMS 버전은 집계 기능만 지원하고 있으며 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시되어야 한다.

STEP 1-2. GROUP BY 절 + ORDER BY 절 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ORDER BY 절을 사용함으로써 부서, 업무별로 정렬이 이루어진다.

```
[예제] SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME, JOB ORDER BY DNAME, JOB;
```

```
[실행 결과] DNAME JOB Total Empl Total Sal ---------- -------- -------- ------- ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDENT 1 5000 RESEARCH ANALYST 2 6000 RESEARCH CLERK 2 1900 RESEARCH MANAGER 1 2975 SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 9개의 행이 선택되었다.
```

STEP 2. ROLLUP 함수 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다.

```
[예제] SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY ROLLUP (DN
```

AME, JOB);

[실행 결과] DNAME JOB Total Empl Total Sal ---------- -------- --------- -------- SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 SALES 6 940 0 RESEARCH CLERK 2 1900 RESEARCH ANALYST 2 6000 RESEARCH MANAGER 1 2975 RESEARCH 5 10875 ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDENT 1 5000 ACCOUNTING 3 8750 14 29025 13개의 행이 선택되었다.

실행 결과에서 2개의 GROUPING COLUMNS(DNAME, JOB)에 대하여 다음과 같은 추가 LEVEL의 집계가 생성된 것을 볼 수 있다.

L1 - GROUP BY 수행시 생성되는 표준 집계 (9건) L2 - DNAME 별 모든 JOB의 SUBTOTAL (3건) L3 - GRAND TOTAL (마지막 행, 1건)

추가로 ROLLUP의 경우 계층 간 집계에 대해서는 LEVEL 별 순서(L1→L2→L3)를 정렬하지만, 계층 내 GROUP BY 수행시 생성되는 표준 집계에는 별도의 정렬을 지원하지 않는다. L1, L2, L3 계층 내 정렬을 위해서는 별도의 ORDER BY 절을 사용해야 한다.

STEP 2-2. ROLLUP 함수 + ORDER BY 절 사용

[예제] 부서명과 업무명을 기준으로 집계한 일반적인 GROUP BY SQL 문장에 ROLLUP 함수를 사용한다. 추가로 ORDER BY 절을 사용해서 부서, 업무별로 정렬한다.

[예제] SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY ROLLUP (DN AME, JOB) ORDER BY DNAME, JOB ;

[실행 결과] DNAME JOB Total Empl Total Sal ---------- ------- -------- -------- ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDE NT 1 5000 ACCOUNTING 3 8750 RESEARCH ANALYST 2 6000 RESEARCH CLERK 2 1900 RESEARCH MANAGER 1 2975 RESEARCH 5 10875 SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 SALES 6 9400 14 29025 13개의 행이 선택되었다.

STEP 3. GROUPING 함수 사용

ROLLUP, CUBE, GROUPING SETS 등 새로운 그룹 함수를 지원하기 위해 GROUPING 함수가 추가되었다.

- ROLLUP이나 CUBE에 의한 소계가 계산된 결과에는 GROUPING(EXPR) = 1 이 표시되고, - 그 외의 결과에는 GROUPING(EXPR) = 0 이 표시된다.

GROUPING 함수와 CASE/DECODE를 이용해, 소계를 나타내는 필드에 원하는 문자열을 지정할 수 있어, 보고서 작성시 유용하게 사용할 수 있다.

[예제] ROLLUP 함수를 추가한 집계 보고서에서 집계 레코드를 구분할 수 있는 GROUPING 함수가 추가된 SQL 문장이다.

[예제] SELECT DNAME, GROUPING(DNAME), JOB, GROUPING(JOB), COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY ROLLUP (DNAME, JOB);

[실행 결과] DNAME ------ GROUPING(DNAME) -------------- JOB --- GROUPING(JOB) ----------- Total Empl -------- Total Sal ------ SALES 0 CLERK 0 1 950 SALES 0 MAN AGER 0 1 2850 SALES 0 SALESMAN 0 4 5600 SALES 0 1 6 9400 RESEARCH 0 CLERK 0 2 1900 RESEARCH 0 ANALYST 0 2 6000 RESEARCH 0 MANAGER 0 1 2975 RESEARCH 0 1 5 10875 ACCOUNTING 0 CLERK 0 1 1300 ACCOUNTING 0 MANAGER 0 1 2450 ACCOUNTING 0 PRESIDENT 0 1 5000 ACCOUNTING 0 1 3 8750 1 1 14 29025 13개의 행이 선택되었다.

부서별, 업무별과 전체 집계를 표시한 레코드에서는 GROUPING 함수가 1을 리턴한 것을 확인할 수 있다. 그리고 전체 합계를 나타내는 결과 라인에서는 부서별 GROUPING 함수와 업무별 GROUPING 함수가 둘 다 1인 것을 알 수 있다.

STEP 4. GROUPING 함수 + CASE 사용

[예제] ROLLUP 함수를 추가한 집계 보고서에서 집계 레코드를 구분할 수 있는 GROUPING 함수와 CASE 함수를 함께 사용한 SQL 문장을 작성한다.

[예제] SELECT CASE GROUPING(DNAME) WHEN 1 THEN 'All Departments' ELSE DNAME END AS DNAME, CASE GROUPING(JOB) WHEN 1 THEN 'All Jobs' ELS E JOB END AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY ROLLUP (DNAME, JO B); Oracle의 경우는 DECODE 함수를 사용해서 좀더 짧게 표현할 수 있다. SELECT DECODE(GROUPING(DNAME), 1, 'All Departments', DNAME) AS DNAME, DECO DE(GROUPING(JOB), 1, 'All Jobs', JOB) AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROU P BY ROLLUP (DNAME, JOB);

[실행 결과] DNAME JOB Total Empl Total Sal ---------- -------- -------- -------- SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 SALES All Jo bs 6 9400 RESEARCH CLERK 2 1900 RESEARCH ANALYST 2 6000 RESEARCH MANAGER 1 2975 RESEARCH All Jobs 5 10875 ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDENT 1 5000 ACCOUNTING All Jobs 3 8750 All Departments All Jobs 14 29025 13개의 행이 선택되었다.

부서별과 전체 집계를 표시한 레코드에서 'ALL JOBS'와 'ALL DEPARTMENTS'라는 사용자 정의 텍스트를 확인할 수 있다. 일부 DBMS는 GROUPING_ID라는 비슷한 용도의 함수를 추가로 사용할 수도 있으므로 참조하기 바란다.

STEP 4-2. ROLLUP 함수 일부 사용

[예제] GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY DNAME, ROLLUP(JOB) 조건으로 변경한 경우이다.

[예제] SELECT CASE GROUPING(DNAME) WHEN 1 THEN 'All Departments' ELSE DNAME END AS DNAME, CASE GROUPING(JOB) WHEN 1 THEN 'All Jobs' ELS E JOB END AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME, ROLLUP(JO B)

[실행 결과] DNAME JOB Total Empl Total Sal ---------- -------- -------- -------- SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 SALES All Jo bs 6 9400 RESEARCH CLERK 2 1900 RESEARCH ANALYST 2 6000 RESEARCH MANAGER 1 2975 RESEARCH All Jobs 5 10875 ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDENT 1 5000 ACCOUNTING All Jobs 3 8750 12개의 행이 선택되었다.

결과는 마지막 ALL DEPARTMENTS & ALL JOBS 줄만 계산이 되지 않았다. ROLLUP이 JOB 칼럼에만 사용되었기 때문에 DNAME에 대한 집계는 필요하지 않기 때문이다.

삭제된 레코드)

All Departments	All Jobs	14	29025

STEP 4-3. ROLLUP 함수 결합 칼럼 사용

[예제] JOB과 MGR는 하나의 집합으로 간주하고, 부서별, JOB & MGR에 대한 ROLLUP 결과를 출력한다.

[예제] SELECT DNAME, JOB, MGR, SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY ROLLUP (DNAME, (JOB, MGR));
☞ JOB, MGR을 소계시 하나의 집합으로 간주하여 구분하지 않음

[실행 결과] DNAME JOB MGR Total Sal --------- --------- ---- ------ SALES CLERK 7698 950 SALES MANAGER 7839 2850 SALES SALESMAN 7698 5600 SALES 9400 RESEARCH CLERK 7788 1100 RESEARCH CLERK 7902 800 RESEARCH ANALYST 7566 6000 RESEARCH MANAGER 7839 2975 RESEARCH 10875 ACCOUNTING CLERK 7782 1300 ACCOUNTING MANAGER 7839 2450 ACCOUNTING PRESIDENT 5000 ACCOUNTING 8750 29025 14개의 행이 선택되었다.

ROLLUP 함수 사용시 괄호로 묶은 JOB과 MGR의 경우 하나의 집합(JOB+MGR) 칼럼으로 간주하여 괄호 내 각 칼럼별 집계를 구하지 않는다.

3. CUBE 함수

ROLLUP에서는 단지 가능한 Subtotal만을 생성하였지만, CUBE는 결합 가능한 모든 값에 대하여 다차원 집계를 생성한다. CUBE를 사용할 경우에는 내부적으로는 Grouping Columns의 순서를 바꾸어서 또 한 번의 Query를 추가 수행해야 한다. 뿐만 아니라 Grand Total은 양쪽의 Query 에서 모두 생성이 되므로 한 번의 Query에서는 제거되어야만 하므로 ROLLUP에 비해 시스템의 연산 대상이 많다. 이처럼 Grouping Columns이 가질 수 있는 모든 경우에 대하여 Subtotal을 생성해야 하는 경우에는 CUBE를 사용하는 것이 바람직하나, ROLLUP에 비해 시스템에 많은 부담을 주므로 사용에 주의해야 한다. CUBE 함수의 경우 표시된 인수들에 대한 계층별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌는 경우 행간에 정렬 순서는 바뀔 수 있어도 데이터 결과는 같다. 그리고 CUBE도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

STEP 5. CUBE 함수 이용

[예제] GROUP BY ROLLUP (DNAME, JOB) 조건에서 GROUP BY CUBE (DNAME, JOB) 조건으로 변경해서 수행한다.

[예제] SELECT CASE GROUPING(DNAME) WHEN 1 THEN 'All Departments' ELSE DNAME END AS DNAME, CASE GROUPING(JOB) WHEN 1 THEN 'All Jobs' ELSE JOB END AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY CUBE (DNAME, JOB) ;

[실행 결과] DNAME JOB Total Empl Total Sal -------------- --------- --------- -------- All Departments All Jobs 14 29025 All Departments CLERK 4 4150 All Departments ANALYST 2 6000 All Departments MANAGER 3 8275 All Departments SALESMAN 4 5600 All Departments PRESIDENT 1 5000 SALES All Jobs 6 9400 SALES CLERK 1 950 SALES MANAGER 1 2850 SALES SALESMAN 4 5600 RESEARCH All Jobs 5 10875 RESEARCH CLERK 2 1900 RESEARCH ANALYST 2 6000 RESEARCH MANAGER 1 2975 ACCOUNTING All Jobs 3 8750 ACCOUNTING CLERK 1 1300 ACCOUNTING MANAGER 1 2450 ACCOUNTING PRESIDENT 1 5000 18개의 행이 선택되었다.

CUBE는 GROUPING COLUMNS이 가질 수 있는 모든 경우의 수에 대하여 Subtotal을 생성하므로 GROUPING COLUMNS의 수가 N이라고 가정하면, 2의 N승 LEVEL의 Subtotal을 생성하게 된다. 실행 결과에서 CUBE 함수 사용으로 ROLLUP 함수의 결과에다 업무별 집계까지 추가해서 출력할 수 있는데, ROLLUP 함수에 비해 업무별 집계를 표시한 5건의 레코드가 추가된 것을 확인할 수 있다. (All Departments - CLERK, ANALYST, MANAGER, SALESMAN, PRESIDENT 별 집계가 5건 추가되었다.)

STEP 5-2. UNION ALL 사용 SQL

UNION ALL은 Set Operation 내용으로, 여러 SQL 문장을 연결하는 역할을 할 수 있다. 위 SQL은 첫 번째 SQL 모듈부터 차례대로 결과가 나오므로 위 CUBE SQL과 결과 데이터는 같으나 행들의 정렬은 다를 수 있다.

[예제] SELECT DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME, JOB UNION ALL SELECT DNAME, 'All Jobs', COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME UNION ALL SELECT 'All Departments', JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY JOB UNION ALL SELECT 'All Departments', 'All Jobs', COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO ;

CUBE 함수를 사용하면서 가장 크게 개선되는 부분은 CUBE 사용 전 SQL에서 EMP, DEPT 테이블을 네 번이나 반복 액세스하는 부분을 CUBE 사용 SQL에서는 한 번으로 줄일 수 있는 부분이다. 기존에 같은 테이블을 네 번 액세스하는 이유가 되었던 부서와 업무별 소계와 총계 부분을 CUBE 함수를 사용함으로써 한 번의 액세스만으로 구현한다. 결과적으로 수행속도 및 자원 사용율을 개선할 수 있으며, SQL 문장도 더 짧아졌으므로 가독성도 높아졌다. 실행 결과는 STEP5의 결과와 동일하다. ROLLUP 함수도 똑 같은 개선 효과를 얻을 수 있다.

4. GROUPING SETS 함수

GROUPING SETS를 이용해 더욱 다양한 소계 집합을 만들 수 있는데, GROUP BY SQL 문장을 여러 번 반복하지 않아도 원하는 결과를 쉽게 얻을 수 있게 되었다. GROUPING SETS에 표시된 인수들에 대한 개별 집계를 구할 수 있으며, 이때 표시된 인수들 간에는 계층 구조인 ROLLUP과는 달리 평등한 관계이므로 인수의 순서가 바뀌어도 결과는 같다. 그리고 GROUPING SETS 함수도 결과에 대한 정렬이 필요한 경우는 ORDER BY 절에 명시적으로 정렬 칼럼이 표시가 되어야 한다.

- 일반 그룹함수를 이용한 SQL

[예제] 일반 그룹함수를 이용하여 부서별, JOB별 인원수와 급여 합을 구하라.

[예제] SELECT DNAME, 'All Jobs' JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY DNAME UNION ALL SELECT 'All Departments' DNAME, JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY JOB ;

[실행 결과] DNAME JOB Total Empl Total Sal ---------- ------- -------- ------ ACCOUNTING All Jobs 3 8750 RESEARCH All Jobs 5 10875 SALES All Jobs 6 9400 All Departments CLERK 4 4150 All Departments SALESMAN 4 5600 All Departments PRESIDENT 1 5000 All Departments MANAGER 3 8275 All Departments ANALYST 2 6000 8개의 행이 선택되었다.

실행 결과는 별도의 ORDER BY 조건을 명시하지 않았기 때문에 DNAME이나 JOB에 대해서 정렬이 되어 있지 않다.

- GROUPING SETS 사용 SQL

[예제] 일반 그룹함수를 GROUPING SETS 함수로 변경하여 부서별, JOB별 인원수와 급여 합을 구하라.

[예제] SELECT DECODE(GROUPING(DNAME), 1, 'All Departments', DNAME) AS DNAME, DECODE(GROUPING(JOB), 1, 'All Jobs', JOB) AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY GROUPING SETS (DNAME, JOB);

[실행 결과] DNAME JOB Total Empl Total Sal ---------- ---------- -------- ------- All Departments CLERK 4 4150 All Departments SALESMAN 4 5600 All Departments PRESIDENT 1 5000 All Departments MANAGER 3 8275 All Departments ANALYST 2 6000 ACCOUNTING All Jobs 3 8750 RESEARCH All Jobs 5 10875 SALES All Jobs 6 9400 8개의 행이 선택되었다.

GROUPING SETS 함수 사용시 UNION ALL을 사용한 일반 그룹함수를 사용한 SQL과 같은 결과를 얻을 수 있으며, 괄호로 묶은 집합 별로(괄호 내는 계층 구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다. GROUPING SETS의 경우 일반 그룹함수를 이용한 SQL과 결과 데이터는 같으나 행들의 정렬 순서는 다를 수 있다.

- GROUPING SETS 사용 SQL - 순서 변경

[예제] 일반 그룹함수를 GROUPING SETS 함수로 변경하여 부서별, JOB별 인원수와 급여 합을 구하는데 GROUPING SETS의 인수들의 순서를 바꾸어 본다.

[예제] SELECT DECODE(GROUPING(DNAME), 1, 'All Departments', DNAME) AS DNAME, DECODE(GROUPING(JOB), 1, 'All Jobs', JOB) AS JOB, COUNT(*) "Total Empl", SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY GROUPING SETS (JOB, DNAME);

[실행 결과] DNAME JOB Total Empl Total Sal ---------- ---------- --------- --------- All Departments CLERK 4 4150 All Departments SALESMAN 4 5600 All Departments PRESIDENT 1 5000 All Departments MANAGER 3 8275 All Departments ANALYST 2 6000 ACCOUNTING All Jobs 3 8750 RESEARCH All Jobs 5 10875 SALES All Jobs 6 9400 8개의 행이 선택되었다.

GROUPING SETS 인수들은 평등한 관계이므로 인수의 순서가 바뀌어도 결과는 같다. (JOB과 DNAME의 순서가 바뀌었지만 결과는 같다.)

- 3개의 인수를 이용한 GROUPING SETS 이용

[예제] 부서-JOB-매니저 별 집계와, 부서-JOB 별 집계와, JOB-매니저 별 집계를 GROUPING SETS 함수를 이용해서 구해본다.

[예제] SELECT DNAME, JOB, MGR, SUM(SAL) "Total Sal" FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO GROUP BY GROUPING SETS ((DNAME, JOB, MGR), (DNAME, JOB), (JOB, MGR)); GROUPING SETS 함수 사용시 괄호로 묶은 집합별로(괄호 내는 계층구조가 아닌 하나의 데이터로 간주함) 집계를 구할 수 있다.

[실행 결과] DNAME JOB MGR Total Sal ---------- ---------- ------- ------- SALES CLERK 7698 950 ACCOUNTING CLERK 7782 1300 RESEARCH CLERK 7788 1100 RESEARCH CLERK 7902 800 RESEARCH ANALYST 7566 6000 SALES MANAGER 7839 2850 RESEARCH MANAGER 7839 2975 ACCOUNTING MANAGER 7839 2450 SALES SALESMAN 7698 5600 ACCOUNTING PRESIDENT 5000 CLERK 7698 950 CLERK 7782 1300 CLERK 7788 1100 CLERK 7902 800 ANALYST 7566 6000 MANAGER 7839 8275 SALESMAN 7698 5600 PRESIDENT 5000 SALES MANAGER 2850 SALES CLERK 950 ACCOUNTING CLERK 1300 ACCOUNTING MANAGER 2450 ACCOUNTING PRESIDENT 5000 RESEARCH MANAGER 2975 SALES SALESMAN 5600 RESEARCH ANALYST 6000 RESEARCH CLERK 1900 27개의 행이 선택되었다.

실행 결과에서 첫 번째 10건의 데이터는 (DNAME+JOB+MGR) 기준의 집계이며, 두 번째 8건의 데이터는 (JOB+MGR) 기준의 집계이며, 세 번째 9건의 데이터는 (DNAME+JOB) 기준의 집계이다.

1. WINDOW FUNCTION 개요

기존 관계형 데이터베이스는 칼럼과 칼럼간의 연산, 비교, 연결이나 집합에 대한 집계는 쉬운 반면, 행과 행간의 관계를 정의하거나, 행과 행간을 비교, 연산하는 것을 하나의 SQL 문으로 처리 하는 것은 매우 어려운 문제였다. PL/SQL, SQL/PL, T-SQL, PRO*C 같은 절차형 프로그램을 작성하거나, INLINE VIEW를 이용해 복잡한 SQL 문을 작성해야 하던 것을 부분적이나마 행과 행간의 관계를 쉽게 정의하기 위해 만든 함수가 바로 WINDOW FUNCTION이다. 윈도우 함수를 활용하면 복잡한 프로그램을 하나의 SQL 문장으로 쉽게 해결할 수 있다. 분석 함수(ANALYTIC FUNCTION)나 순위 함수(RANK FUNCTION)로도 알려져 있는 윈도우 함수(ANSI/ISO SQL 표준은 WINDOW FUNCTION이란 용어를 사용함)는 데이터웨어하우스에서 발전한 기능이다. SQL 사용자 입장에서는 INLINE VIEW 이후 SQL의 중요한 기능이 추가되었다고 할 수 있으며, 많은 프로그램이나 튜닝 팁을 대체할 수 있을 것이다. 복잡하거나 자원을 많이 사용하는 튜닝 기법들을 대체할 수 있는 DBMS의 새로운 기능은 튜닝 관점에서도 최적화된 방법이므로 적극적으로 활용할 필요가 있다. 같은 결과가 나오는 변형된 튜닝 문장보다는 DBMS 벤더에서 최적화된 자원을 사용하도록 만들어진 새로운 기능을 사용하는 것이 일반적으로 더욱 효과가 좋기 때문이다. WINDOW 함수는 기존에 사용하던 집계 함수도 있고, 새로이 WINDOW 함수 전용으로 만들어진 기능도 있다. 그리고 WINDOW 함수는 다른 함수와는 달리 중첩(NEST)해서 사용하지는 못하지만, 서브쿼리에서는 사용할 수 있다.

- WINDOW FUNCTION 종류

WINDOW FUNCTION의 종류는 크게 다섯 개의 그룹으로 분류할 수 있는데 벤더별로 지원하는 함수에는 차이가 있다. 첫 번째, 그룹 내 순위(RANK) 관련 함수는 RANK, DENSE_RANK, ROW_NUMBER 함수가 있다. ANSI/ISO SQL 표준과 Oracle, SQL Server 등 대부분의 DBMS에서 지원하고 있다. 두 번째, 그룹 내 집계(AGGREGATE) 관련 함수는 일반적으로 많이 사용하는 SUM, MAX, MIN, AVG, COUNT 함수가 있다. ANSI/ISO SQL 표준과 Oracle, SQL Server 등 대부분의 DBMS에서 지원하고 있는데, SQL Server의 경우 집계 함수는 뒤에서 설명할 OVER 절 내의 ORDER BY 구문을 지원하지 않는다. 세 번째, 그룹 내 행 순서 관련 함수는 FIRST_VALUE, LAST_VALUE, LAG, LEAD 함수가 있다. Oracle에서만 지원되는 함수이기는 하지만, FIRST_VALUE, LAST_VALUE 함수는 MAX, MIN 함수와 비슷한 결과를 얻을 수 있고, LAG, LEAD 함수는 DW에서 유용하게 사용되는 기능이므로 같이 설명하도록 한다. 네 번째, 그룹 내 비율 관련 함수는 CUME_DIST, PERCENT_RANK, NTILE, RATIO_TO_REPORT 함수가 있다. CUME_DIST, PERCENT_RANK 함수는 ANSI/ISO SQL 표준과 Oracle DBMS에서 지원하고 있으며, NTILE 함수는 ANSI/ISO SQL 표준에는 없지만, Oracle, SQL Server에서 지원하고 있다. 마지막으로 RATIO_TO_REPORT 함수는 Oracle에서만 지원되는 함수이기는 하나, 현업에서 유용한 기능을 구현하는데 참조하기 위해 설명하도록 한다. 다섯 번째, 선형 분석을 포함한 통계 분석 관련 함수가 있는데, 통계에 특화된 기능이므로 본 가이드에서는 설명을 생략한다. 아래는 Oracle의 통계 관련 함수를 참조로 표시한 것이다.

CORR, COVAR_POP, COVAR_SAMP, STDDEV, STDDEV_POP, STDDEV_SAMP, VARIANCE, VAR_POP, VAR_SAMP, REGR_(LINEAR REGRESSION), REGR_SLOPE, REGR_INTERCEPT, REGR_COUNT, REGR_R2, REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SYY, REGR_SXY

- WINDOW FUNCTION SYNTAX

- WINDOW 함수에는 OVER 문구가 키워드로 필수 포함된다.

```
SELECT WINDOW_FUNCTION (ARGUMENTS) OVER ( [PARTITION BY 칼럼] [ORDER BY 절] [WINDOWING 절] ) FROM 테이블 명;
```

- WINDOW_FUNCTION : 기존에 사용하던 함수도 있고, 새롭게 WINDOW 함수용으로 추가된 함수도 있다. - ARGUMENTS (인수) : 함수에 따라 0 ~ N개의 인수가 지정될 수 있다. - PARTITION BY 절 : 전체 집합을 기준에 의해 소그룹으로 나눌 수 있다. - ORDER BY 절 : 어떤 항목에 대해 순위를 지정할 지 ORDER BY 절을 기술한다. - WINDOWING 절 : WINDOWING 절은 함수의 대상이 되는 행 기준의 범위를 강력하게 지정할 수 있다. ROWS는 물리적인 결과 행의 수를, RANGE는 논리적인 값에 의한 범위를 나타내는데, 둘 중의 하나를 선택해서 사용할 수 있다. 다만, WINDOWING 절은 SQL Server에서는 지원하지 않는다.

```
BETWEEN 사용 타입 ROWS | RANGE BETWEEN UNBOUNDED PRECEDING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING AND UNBOUNDED FOLLOWING | CURRENT ROW | VALUE_EXPR PRECEDING/FOLLOWING BETWEEN 미사용 타입 ROWS | RANGE UNBOUNDED PRECEDING | CURRENT ROW | VALUE_EXPR PRECEDING
```

2. 그룹 내 순위 함수

가. RANK 함수

RANK 함수는 ORDER BY를 포함한 QUERY 문에서 특정 항목(칼럼)에 대한 순위를 구하는 함수이다. 이때 특정 범위(PARTITION) 내에서 순위를 구할 수도 있고 전체 데이터에 대한 순위를 구할 수도 있다. 또한 동일한 값에 대해서는 동일한 순위를 부여하게 된다.

[예제] 사원 데이터에서 급여가 높은 순서와 JOB 별로 급여가 높은 순서를 같이 출력한다.

```
[예제] SELECT JOB, ENAME, SAL, RANK( ) OVER (ORDER BY SAL DESC) ALL_RANK, RANK( ) OVER (PARTITION BY JOB ORDER BY SAL DESC) JOB_RANK FROM EMP;
```

```
[실행 결과] JOB ENAME SAL ALL_RANK JOB_RANK --------- ----- ---- --------- ------- PRESIDENT KING 5000 1 1 ANALYST FORD 3000 2 1 ANALYST SCOTT 3000 2 1 MANAGER JONES 2975 4 1 MANAGER BLAKE 2850 5 2 MANAGER CLARK 2450 6 3 SALESMAN ALLEN 1600 7 1 SALESMAN TURNER 1500 8 2 CLERK MILLE
```

R 1300 9 1 SALESMAN WARD 1250 10 3 SALESMAN MARTIN 1250 10 3 CLERK ADAMS 1100 12 2 CLERK JAMES 950 13 3 CLERK SMITH 800 14 4 13개의 행이 선택되었다.

업무 구분이 없는 ALL_RANK 칼럼에서 FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 같은 순위를 부여한다. 그리고 업무를 PARTITION으로 구분한 JOB_RANK의 경우 같은 업무 내 범위에서만 순위를 부여한다. 하나의 SQL 문장에 ORDER BY SAL DESC 조건과 PARTITION BY JOB 조건이 충돌이 났기 때문에 JOB 별로는 정렬이 되지 않고, ORDER BY SAL DESC 조건으로 정렬이 되었다.

[예제] 앞의 SQL문의 결과는 JOB과 SALARY 기준으로 정렬이 되어있지 않다. 새로운 SQL에서는 전체 SALARY 순위를 구하는 ALL_RANK 칼럼은 제외하고, 업무별로 SALARY 순서를 구하는 JOB_RANK만 알아보도록 한다.

[예제] SELECT JOB, ENAME, SAL, RANK() OVER (PARTITION BY JOB ORDER BY SAL DESC) JOB_RANK FROM EMP;

[실행 결과] JOB ENAME SAL JOB_RANK --------- ------ ----- ------- ANALYST FORD 3000 1 ANALYST SCOTT 3000 1 CLERK MILLER 1300 1 CLERK ADAMS 1100 2 CLERK JAMES 950 3 CLERK SMITH 800 4 MANAGER JONES 2975 1 MANAGER BLAKE 2850 2 MANAGER CLARK 2450 3 PRESIDENT KING 5000 1 SALESMAN ALLEN 1600 1 SALESMAN TURNER 1500 2 SALESMAN MARTIN 1250 3 SALESMAN WARD 1250 3 13개의 행이 선택되었다.

업무별로 SALARY 순서를 구하는 JOB_RANK만 사용한 경우 파티션의 기준이 된 JOB과 SALARY 별로 정렬이 되어 있는 것을 알 수 있다.

나. DENSE_RANK 함수

DENSE_RANK 함수는 RANK 함수와 흡사하나, 동일한 순위를 하나의 건수로 취급하는 것이 틀린 점이다.

[예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 하나의 등수로 간주한 결과도 같이 출력한다.

[예제] SELECT JOB, ENAME, SAL, RANK( ) OVER (ORDER BY SAL DESC) RANK, DENSE_RANK( ) OVER (ORDER BY SAL DESC) DENSE_RANK FROM EMP;

[실행 결과] JOB ENAME SAL RANK DENSE_RANK ---------- ------ ---- ---- --------- PRESIDENT KING 5000 1 1 ANALYST FORD 3000 2 2 ANALYST SCOTT 3000 2 2 MANAGER JONES 2975 4 3 MANAGER BLAKE 2850 5 4 MANAGER CLARK 2450 6 5 SALESMAN ALLEN 1600 7 6 SALESMAN TURNER 1500 8 7 CLERK MILLER 1300 9 8 SALESMAN WARD 1250 10 9 SALESMAN MARTIN 1250 10 9 CLERK ADAMS 1100 12 10 CLERK JAMES 950 13 11 CLERK SMITH 800 14 12 13개의 행이 선택되었다.

FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 RANK와 DENSE_RANK 칼럼에서 모두 같은 순위를 부여한다. 그러나 RANK와 DENSE_RANK의 차이를 알 수 있는 데이터는 FORD와 SCOTT의 다음 순위인 JONES의 경우 RANK는 4등으로 DENSE_RANK는 3등으로 표시되어 있다. 마찬가지로 WARD와 MARTIN의 다음 순위인 ADAMS의 경우 RANK는 12등으로 DENSE_RANK는 10등으로 표시되어 있다.

다. ROW_NUMBER 함수

ROW_NUMBER 함수는 RANK나 DENSE_RANK 함수가 동일한 값에 대해서는 동일한 순위를 부여하는데 반해, 동일한 값이라도 고유한 순위를 부여한다.

[예제] 사원데이터에서 급여가 높은 순서와, 동일한 순위를 인정하지 않는 등수도 같이 출력한다.

[예제] SELECT JOB, ENAME, SAL, RANK( ) OVER (ORDER BY SAL DESC) RANK, ROW_NUMBER() OVER (ORDER BY SAL DESC) ROW_NUMBER FROM EMP;

[실행 결과] JOB ENAME SAL RANK ROW_NUMBER --------- ------ ----- ----- ---------- PRESIDENT KING 5000 1 1 ANALYST FORD 3000 2 2 ANALYST SCOTT 3000 2 3 MANAGER JONES 2975 4 4 MANAGER BLAKE 2850 5 5 MANAGER CLARK 2450 6 6 SALESMAN ALLEN 1600 7 7 SALESMAN TURNER 1500 8 8 CLERK MILLER 1300 9 9 SALESMAN WARD 1250 10 10 SALESMAN MARTIN 1250 10 11 CLERK ADAMS 1100 12 12 CLERK JAMES 950 13 13 CLERK SMITH 800 14 14 14개의 행이 선택되었다.

FORD와 SCOTT, WARD와 MARTIN은 동일한 SALARY이므로 RANK는 같은 순위를 부여했지만, ROW_NUMBER의 경우 동일한 순위를 배제하기 위해 유니크한 순위를 정한다. 위 경우는 같은 SALARY에서는 어떤 순서가 정해질지 알 수 없다. (Oracle의 경우 rowid가 적은 행이 먼저 나온다) 이 부분은 데이터베이스 별로 틀린 결과가 나올 수 있으므로, 만일 동일 값에 대한 순서까지 관리하고 싶으면 ROW_NUMBER( ) OVER (ORDER BY SAL DESC, ENAME) 같이 ORDER BY 절을 이용해 추가적인 정렬 기준을 정의해야 한다.

3. 일반 집계 함수

가. SUM 함수

SUM 함수를 이용해 파티션별 윈도우의 합을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들의 SALARY 합을 구한다.

[예제] SELECT MGR, ENAME, SAL, SUM(SAL) OVER (PARTITION BY MGR) MGR_SUM FROM EMP; PARTITION BY MGR 구문을 통해 매니저별로 데이터를 파티션화 한다.

[실행 결과] MGR ENAME SAL MGR_SUM ---- ------ ---- ------- 7566 FORD 3000 6000 7566 SCOTT 3000 6000 7698 JAMES 950 6550 7698 ALLEN 1600 6550 7698 WARD 1250 6550 7698 TURNER 1500 6550 7698 MARTIN 1250 6550 7782 MILLER 1300 1300 7788 ADAMS 1100 1100 7839 BLAKE 2850 8275 7839 JONES 2975 8275 7839 CLARK 2450 8275 7902 SMITH 800 800 KING 5000 5000 14개의 행이 선택되었다.

[예제] OVER 절 내에 ORDER BY 절을 추가해 파티션 내 데이터를 정렬하고 이전 SALARY 데이터까지의 누적값을 출력한다. (SQL Server의 경우 집계 함수의 경우 OVER 절 내의 ORDER BY 절을 지원하지 않는다.)

[예제] SELECT MGR, ENAME, SAL, SUM(SAL) OVER (PARTITION BY MGR ORDER BY SAL RANGE UNBOUNDED PRECEDING) as MGR_SUM FROM EMP RANGE UNBOUNDED PRECEDING : 현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.

[실행 결과] MGR ENAME SAL MGR_SUM ---- -------- ---- ------- 7566 SCOTT 3000 6000 7566 FORD 3000 6000 7698 JAMES 950 950 7698 WARD * 1250 3450 7698

MARTIN * 1250 3450 7698 TURNER 1500 4950 7698 ALLEN 1600 6550 7782 MILLER 1300 1300 7788 ADAMS 1100 1100 7839 CLARK 2450 2450 7839 BLAKE 2850 5300 7839 JONES 2975 8275 7902 SMITH 800 800 KING 5000 5000 14개의 행이 선택되었다.

* 표시된 7699-WARD와 7699-MARTIN의 급여가 같으므로, 같은 ORDER로 취급하여 950+1250+1250=3450의 값이 되었다. 7698-TURNER의 경우 950+1250+1250+1500=4950의 누적합을 가진다.

나. MAX 함수

MAX 함수를 이용해 파티션별 윈도우의 최대값을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들의 SALARY 중 최대값을 같이 구한다.

[예제] SELECT MGR, ENAME, SAL, MAX(SAL) OVER (PARTITION BY MGR) as MGR_MAX FROM EMP;

[실행 결과] MGR ENAME SAL MGR_MAX ---- ----- ---- ------- 7566 FORD 3000 3000 7566 SCOTT 3000 3000 7698 JAMES 950 1600 7698 ALLEN 1600 1600 7698 WARD 1250 1600 7698 TURNER 1500 1600 7698 MARTIN 1250 1600 7782 MILLER 1300 1300 7788 ADAMS 1100 1100 7839 BLAKE 2850 2975 7839 JONES 2975 2975 7839 CLARK 2450 2975 7902 SMITH 800 800 KING 5000 5000 14개의 행이 선택되었다.

실행 결과를 확인하면 파티션 내의 최대값을 파티션 내 모든 행에서 MGR_MAX라는 칼럼 값으로 가질 수 있다.

[예제] 추가로, INLINE VIEW를 이용해 파티션별 최대값을 가진 행만 추출할 수도 있다.

[예제] SELECT MGR, ENAME, SAL FROM (SELECT MGR, ENAME, SAL, MAX(SAL) OVER (PARTITION BY MGR) as IV_MAX_SAL FROM EMP) WHERE SAL = IV_MAX_SAL ;

[실행 결과] MGR ENAME SAL ------ ------- ----- 7566 FORD 3000 7566 SCOTT 3000 7698 ALLEN 1600 7782 MILLER 1300 7788 ADAMS 1100 7839 JONES 2975 7902 SMITH 800 KING 5000 8개의 행이 선택되었다.

실행 결과를 보면 MGR 7566의 SCOTT, FORD는 같은 최대값을 가지므로, WHERE SAL = IV_MAX_SAL 조건에 의해 두건 모두 추출되었다.

다. MIN 함수

MIN 함수를 이용해 파티션별 윈도우의 최소값을 구할 수 있다.

[예제] 사원들의 급여와 같은 매니저를 두고 있는 사원들을 입사일자를 기준으로 정렬하고, SALARY 최소값을 같이 구한다.

[예제] SELECT MGR, ENAME, HIREDATE, SAL, MIN(SAL) OVER(PARTITION BY MGR ORDER BY HIREDATE) as MGR_MIN FROM EMP;

[실행 결과] MGR ENAME HIREDATE SAL MGR_MIN ---- ------ --------- ---- ------- 7566 FORD 1981-12-03 3000 3000 7566 SCOTT 1987-07-13 3000 3000 7698 ALLEN 1981-02-20 1600 1600 7698 WARD 1981-02-22 1250 1250 7698 TURNER 1981-09-08 1500 1250 7698 MARTIN 1981-09-28 1250 1250 7698 JAMES 1981-12-03 950 950 7782 MILLER 1982-01-23 1300 1300 7788 ADAMS 1987-07-13 1100 1100 7839 JONES 1981-04-02 2975 2975 7839 BLAKE 1981-05-01 2850 2850 7839 CLARK 1981-06-09 2450 2450 7902 SMITH 1980-12-17 800 800 KING 1981-11-17 5000 5000 14개의 행이 선택되었다.

라. AVG 함수

AVG 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계값을 구할 수 있다.

[예제] EMP 테이블에서 같은 매니저를 두고 있는 사원들의 평균 SALARY를 구하는데, 조건은 같은 매니저 내에서 자기 바로 앞의 사번과 바로 뒤의 사번인 직원만을 대상으로 한다.

[예제] SELECT MGR, ENAME, HIREDATE, SAL, ROUND (AVG(SAL) OVER (PARTITION BY MGR ORDER BY HIREDATE ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)) as MGR_AVG FROM EMP; ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING : 현재 행을 기준으로 파티션 내에서 앞의 한 건, 현재 행, 뒤의 한 건을 범위로 지정한다. (ROWS는 현재 행의 앞뒤 건수를 말하는 것임)

[실행 결과] MGR ENAME HIREDATE SAL MGR_AVG ---- ------ -------- ---- ------- 7566 FORD 1981-12-03 3000 3000 7566 SCOTT 1987-07-13 3000 3000 7698 ALLEN 1981-02-20 1600 1425 7698 WARD 1981-02-22 1250 1450 7698 TURNER 1981-09-08 1500 1333 7698 MARTIN 1981-09-28 1250 1233 7698 JAMES 1981-12-03 950 1100 7782 MILLER 1982-01-23 1300 1300 7788 ADAMS 1987-07-13 1100 1100 7839 JONES 1981-04-02 2975 2913 7839 BLAKE 1981-05-01 2850 2758 7839 CLARK 1981-06-09 2450 2650 7902 SMITH 1980-12-17 800 800 KING 1981-11-17 5000 5000 14개의 행이 선택되었다.

실행 결과에서 ALLEN의 경우 파티션 내에서 첫 번째 데이터이므로 앞의 한 건은 평균값 집계 대상이 없다. 결과적으로 평균값 집계 대상은 본인의 데이터와 뒤의 한 건으로 평균값을 구한다. (1600 + 1250) / 2 = 1425의 값을 가진다. TURNER의 경우 앞의 한건과, 본인의 데이터와, 뒤의 한 건으로 평균값을 구한다. (1250 + 1500 + 1250) / 3 = 1333의 값을 가진다. JAMES의 경우 파티션 내에서 마지막 데이터이므로 뒤의 한 건을 제외한, 앞의 한 건과 본인의 데이터를 가지고 평균값을 구한다. (1250 + 950) / 2 = 1100의 값을 가진다.

마. COUNT 함수

COUNT 함수와 파티션별 ROWS 윈도우를 이용해 원하는 조건에 맞는 데이터에 대한 통계값을 구할 수 있다.

[예제] 사원들을 급여 기준으로 정렬하고, 본인의 급여보다 50 이하가 적거나 150 이하로 많은 급여를 받는 인원수를 출력하라.

[예제] SELECT ENAME, SAL, COUNT(*) OVER (ORDER BY SAL RANGE BETWEEN 50 PRECEDING AND 150 FOLLOWING) as SIM_CNT FROM EMP; RANGE BETWEEN 50 PRECEDING AND 150 FOLLOWING : 현재 행의 급여값을 기준으로 급여가 -50에서 +150의 범위 내에 포함된 모든 행이 대상이 된다. (RANGE는 현재 행의 데이터 값을 기준으로 앞뒤 데이터 값의 범위를 표시하는 것임)

[실행 결과] ENAME SAL SIM_CNT (범위값) ------ ---- ------ --------- SMITH 800 2 (750~ 950) JAMES 950 2 (900~1100) ADAMS ** 1100 3 (1050~1250) WARD 1250 3 (1200~1400) MARTIN 1250 3 (1200~1400) MILLER 1300 3 (1250~1450) TURNER 1500 2 (1450~1650) ALLEN 1600 1 (1550~1750) CLARK 2450 1 (2400~2600) BLAKE 2850 4 (2800~3000) JONES 2975 3 (2925~3125) SCOTT 3000 3 (2950~3100) FORD 3000 3 (2950~3100) KING 5000 1 (4950~5100) 14개의 행이 선택되었다.

위 SQL 문장은 파티션이 지정되지 않았으므로 모든 건수를 대상으로 -50 ~ +150 기준에 맞는지 검사하게 된다. ORDER BY SAL로 정렬이 되어 있으므로 비교 연산이 쉬워진다. ** 표시된 ADAMS의 경우 자기가 가지고 있는 SALARY 1100을 기준으로 -50에서 +150까지 값을 가진 1050에서 1250까지의 값을 가진 JAMES(950), ADAMS(1100), WARD(1250) 3명의 데이터 건수를 구할 수 있다.

4. 그룹 내 행 순서 함수

가. FIRST_VALUE 함수

FIRST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 먼저 나온 값을 구한다. SQL Server에서는 지원하지 않는 함수이다. MIN 함수를 활용하여 같은 결과를 얻을 수도 있다.

[예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 먼저 나온 값을 출력한다.

[예제] SELECT DEPTNO, ENAME, SAL, FIRST_VALUE(ENAME) OVER (PARTITION BY DEPTNO ORDER BY SAL DESC ROWS UNBOUNDED PRECEDING) as DEPT_RICH FROM EMP; RANGE UNBOUNDED PRECEDING : 현재 행을 기준으로 파티션 내의 첫 번째 행까지의 범위를 지정한다.

[실행 결과] DEPTNO ENAME SAL DEPT_RICH ------ ------- ---- -------- 10 KING 5000 KING 10 CLARK 2450 KING 10 MILLER 1300 KING 20 SCOTT * 3000 SCOTT 20 FORD * 3000 SCOTT 20 JONES 2975 SCOTT 20 ADAMS 1100 SCOTT 20 SMITH 800 SCOTT 30 BLAKE 2850 BLAKE 30 ALLEN 1600 BLAKE 30 TURNER 1500 BLAKE 30 MARTIN 1250 BLAKE 30 WARD 1250 BLAKE 30 JAMES 950 BLAKE 14개의 행이 선택되었다.

실행 결과를 보면 같은 부서 내에 최고 급여를 받는 사람이 둘 있는 경우, 즉, * 표시가 있는 부서번호 20의 SCOTT과 FORD 중에서 어느 사람이 최고 급여자로 선택될지는 위의 SQL 문만 가지고는 판단할 수 없다. FIRST_VALUE는 다른 함수와 달리 공동 등수를 인정하지 않고 처음 나온 행만을 처리한다. 위처럼 공동 등수가 있을 경우에 의도적으로 세부 항목을 정렬하고 싶다면 별도의 정렬 조건을 가진 INLINE VIEW를 사용하거나, OVER () 내의 ORDER BY 절에 칼럼을 추가해야 한다.

[예제] 앞의 SQL 문장에서 같은 값을 가진 FIRST_VALUE를 처리하기 위해 ORDER BY 정렬 조건을 추가한다.

[예제] SELECT DEPTNO, ENAME, SAL, FIRST_VALUE(ENAME) OVER (PARTITION BY DEPTNO ORDER BY SAL DESC, ENAME ASC ROWS UNBOUNDED PRECEDING) as RICH_EMP FROM EMP;

[실행 결과] DEPTNO ENAME SAL RICH_EMP ------ ------- ---- ------- 10 KING 5000 KING 10 CLARK 2450 KING 10 MILLER 1300 KING 20 FORD 3000 FORD 20 SCOTT 3000 FORD 20 JONES 2975 FORD 20 ADAMS 1100 FORD 20 SMITH 800 FORD 30 BLAKE 2850 BLAKE 30 ALLEN 1600 BLAKE 30 TURNER 1500 BLAKE 30 MARTIN 1250 BLAKE 30 WARD 1250 BLAKE 30 JAMES 950 BLAKE 14개의 행이 선택되었다.

SQL에서 같은 부서 내에 최고 급여를 받는 사람이 둘 있는 경우를 대비해서 이름을 두 번째 정렬 조건으로 추가한다. 실행 결과를 확인하면 부서번호 20의 최고 급여자가 이전의 SCOTT 값에서 ASCII 코드가 적은 값인 FORD로 변경 된 것을 확인할 수 있다.

나. LAST_VALUE 함수

LAST_VALUE 함수를 이용해 파티션별 윈도우에서 가장 나중에 나온 값을 구한다. SQL Server에서는 지원하지 않는 함수이다. MAX 함수를 활용하여 같은 결과를 얻을 수도 있다.

[예제] 부서별 직원들을 연봉이 높은 순서부터 정렬하고, 파티션 내에서 가장 마지막에 나온 값을 출력한다.

[예제] SELECT DEPTNO, ENAME, SAL, LAST_VALUE(ENAME) OVER (PARTITION BY DEPTNO ORDER BY SAL DESC ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as DEPT_POOR FROM EMP; ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING: 현재 행을 포함해서 파티션 내의 마지막 행까지의 범위를 지정한다.

[실행 결과] DEPTNO ENAME SAL DEPT_POOR ------ ------- ---- --------- 10 KING 5000 MILLER 10 CLARK 2450 MILLER 10 MILLER 1300 MILLER 20 SCOTT 3000 SMITH 20 FORD 3000 SMITH 20 JONES 2975 SMITH 20 ADAMS 1100 SMITH 20 SMITH 800 SMITH 30 BLAKE 2850 JAMES 30 ALLEN 1600 JAMES 30 TURNER 1500 JAMES 30 MARTIN 1250 JAMES 30 WARD 1250 JAMES 30 JAMES 950 JAMES 14개의 행이 선택되었다.

실행 결과에서 LAST_VALUE는 다른 함수와 달리 공동 등수를 인정하지 않고 가장 나중에 나온 행만을 처리한다. 만일 공동 등수가 있을 경우를 의도적으로 정렬하고 싶다면 별도의 정렬 조건을 가진 INLINE VIEW를 사용하거나, OVER () 내의 ORDER BY 조건에 칼럼을 추가해야 한다.

다. LAG 함수

LAG 함수를 이용해 파티션별 윈도우에서 이전 몇 번째 행의 값을 가져올 수 있다. SQL Server에서는 지원하지 않는 함수이다.

[예제] 직원들을 입사일자가 빠른 기준으로 정렬을 하고, 본인보다 입사일자가 한 명 앞선 사원의 급여를 본인의 급여와 함께 출력한다.

[예제] SELECT ENAME, HIREDATE, SAL, LAG(SAL) OVER (ORDER BY HIREDATE) as PREV_SAL FROM EMP WHERE JOB = 'SALESMAN' ;

[실행 결과] ENAME HIREDATE SAL PREV_SAL ------- --------- ---- ------- ALLEN 1981-02-20 1600 WARD 1981-02-22 1250 1600 TURNER 1981-09-08 1500 1250 MARTIN 1981-09-28 1250 1500 4개의 행이 선택되었다.

[예제] LAG 함수는 3개의 ARGUMENTS 까지 사용할 수 있는데, 두 번째 인자는 몇 번째 앞의 행을 가져올지 결정하는 것이고 (DEFAULT 1), 세 번째 인자는 예를 들어 파티션의 첫 번째 행의 경우 가져올 데이터가 없어 NULL 값이 들어오는데 이 경우 다른 값으로 바꾸어 줄 수 있다. 결과적으로 NVL이나 ISNULL 기능과 같다.

[예제] SELECT ENAME, HIREDATE, SAL, LAG(SAL, 2, 0) OVER (ORDER BY HIREDATE) as PREV_SAL FROM EMP WHERE JOB = 'SALESMAN' LAG(SAL, 2, 0)의 기능은 두 행 앞의 SALARY를 가져오고, 가져올 값이 없는 경우는 0으로 처리한다.

[실행 결과] ENAME HIREDATE SAL PREV_SAL ------- --------- ---- ------- ALLEN 1981-02-20 1600 0 WARD 1981-02-22 1250 0 TURNER 1981-09-08 1500 1600 MARTIN 1981-09-28 1250 1250 4개의 행이 선택되었다.

라. LEAD 함수

LEAD 함수를 이용해 파티션별 윈도우에서 이후 몇 번째 행의 값을 가져올 수 있다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 직원들을 입사일자가 빠른 기준으로 정렬을 하고, 바로 다음에 입사한 인력의 입사일자를 함께 출력한다.

```
[예제] SELECT ENAME, HIREDATE, LEAD(HIREDATE, 1) OVER (ORDER BY HIREDATE) as "NEXTHIRED" FROM EMP;
```

```
[실행 결과] ENAME HIREDATE NEXTHIRED -------- --------- --------- ALLEN 1981-02-20 1981-02-22 WARD 1981-02-22 1981-04-02 TURNER 1981-09-08 1981-09-28 MARTIN 1981-09-28 4개의 행이 선택되었다.
```

LEAD 함수는 3개의 ARGUMENTS 까지 사용할 수 있는데, 두 번째 인자는 몇 번째 후의 행을 가져올지 결정하는 것이고 (DEFAULT 1), 세 번째 인자는 예를 들어 파티션의 마지막 행의 경우 가져올 데이터가 없어 NULL 값이 들어오는데 이 경우 다른 값으로 바꾸어 줄 수 있다. 결과적으로 NVL이나 ISNULL 기능과 같다.

5. 그룹 내 비율 함수

가. RATIO_TO_REPORT 함수

RATIO_TO_REPORT 함수를 이용해 파티션 내 전체 SUM(칼럼)값에 대한 행별 칼럼 값의 백분율을 소수점으로 구할 수 있다. 결과 값은 > 0 & <= 1 의 범위를 가진다. 그리고 개별 RATIO의 합을 구하면 1이 된다. SQL Server에서는 지원하지 않는 함수이다.

[예제] JOB이 SALESMAN인 사원들을 대상으로 전체 급여에서 본인이 차지하는 비율을 출력한다.

```
[예제] SELECT ENAME, SAL, ROUND(RATIO_TO_REPORT(SAL) OVER (), 2) as R_R FROM EMP WHERE JOB = 'SALESMAN';
```

```
[실행 결과] ENAME SAL R_R ------ ---- ---- ALLEN 1600 0.29 (1600 / 5600) WARD 1250 0.22 (1250 / 5600) MARTIN 1250 0.22 (1250 / 5600) TURNER 1500 0.27 500 / 5600) 4개의 행이 선택되었다.
```

실행 결과에서 전체 값은 1650 + 1250 + 1250 + 1500 = 5600이 되고, RATIO_TO_REPORT 함수 연산의 분모로 사용된다. 그리고 개별 RATIO의 전체 합을 구하면 1이 되는 것을 확인할 수 있다.

```
0.29 + 0.22 + 0.22 + 0.27 = 1
```

나. PERCENT_RANK 함수

PERCENT_RANK 함수를 이용해 파티션별 윈도우에서 제일 먼저 나오는 것을 0으로, 제일 늦게 나오는 것을 1로 하여, 값이 아닌 행의 순서별 백분율을 구한다. 결과 값은 >= 0 & <= 1 의 범위를 가진다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 같은 부서 소속 사원들의 집합에서 본인의 급여가 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

```
[예제] SELECT DEPTNO, ENAME, SAL, PERCENT_RANK() OVER (PARTITION BY DEPTNO ORDER BY SAL DESC) as P_R FROM EMP;
```

```
[실행 결과] DEPTNO ENAME SAL P_R ------ ------ ---- ---- 10 KING 5000 0 10 CLARK 2450 0.5 10 MILLER 1300 1 20 SCOTT 3000 0 20 FORD 3000 0 20 JONES 2975 0.5 20 ADAMS 1100 0.75 20 SMITH 800 1 30 BLAKE 2850 0 30 ALLEN 1600 0.2 30 TURNER 1500 0.4 30 MARTIN 1250 0.6 30 WARD 1250 0.6 30 JAMES 950 1 14개의 행이 선택되었다.
```

DEPTNO 10의 경우 3건이므로 구간은 2개가 된다. 0과 1 사이를 2개의 구간으로 나누면 0, 0.5, 1이 된다. DEPTNO 20의 경우 5건이므로 구간은 4개가 된다. 0과 1 사이를 4개의 구간으로 나누면 0, 0.25, 0.5, 0.75, 1이 된다. DEPTNO 30의 경우 6건이므로 구간은 5개가 된다. 0과 1 사이를 5개의 구간으로 나누면 0, 0.2, 0.4, 0.6, 0.8, 1이 된다. SCOTT, FORD와 WARD, MARTIN의 경우 ORDER BY SAL DESC 구문에 의해 급여가 같으므로 같은 ORDER로 취급한다.

다. CUME_DIST 함수

CUME_DIST 함수를 이용해 파티션별 윈도우의 전체건수에서 현재 행보다 작거나 같은 건수에 대한 누적백분율을 구한다. 결과 값은 > 0 & <= 1 의 범위를 가진다. 참고로 SQL Server에서는 지원하지 않는 함수이다.

[예제] 같은 부서 소속 사원들의 집합에서 본인의 급여가 누적 순서상 몇 번째 위치쯤에 있는지 0과 1 사이의 값으로 출력한다.

```
[예제] SELECT DEPTNO, ENAME, SAL, CUME_DIST() OVER (PARTITION BY DEPTNO ORDER BY SAL DESC) as CUME_DIST FROM EMP;
```

```
[실행 결과] DEPTNO ENAME SAL CUME_DIST ------ ------ ---- -------- 10 KING 5000 0.3333 10 CLARK 2450 0.6667 10 MILLER 1300 1.0000 20 SCOTT * 3000 0.4000 20 FORD * 3000 0.4000 20 JONES 2975 0.6000 20 ADAMS 1100 0.8000 20 SMITH 800 1.0000 30 BLAKE 2850 0.1667 30 ALLEN 1600 0.3333 30 TURNER 1500 0.5000 30 MARTIN ** 1250 0.8333 30 WARD ** 1250 0.8333 30 JAMES 950 1.0000 14개의 행이 선택되었다.
```

DEPTNO가 10인 경우 윈도우가 전체 3건이므로 0.3333 단위의 간격을 가진다. 즉, 0.3333, 0.6667, 1의 값이 된다. DEPTNO가 20인 경우 윈도우가 전체 5건이므로 0.2000 단위의 간격을 가진다. 즉, 0.2000, 0.4000, 0.6000, 0.8000, 1의 값이 된다. DEPTNO가 30인 경우 윈도우가 전체 6건이므로 0.1667 단위의 간격을 가진다. 즉, 0.1667, 0.3333, 0.5000, 0.6667, 0.8333, 1의 값이 된다. * 표시가 있는 SCOTT, FORD와 ** 표시가 있는 WARD, MARTIN의 경우 ORDER BY SAL에 의해 SAL 이 같으므로 같은 ORDER로 취급한다. 다른 WINDOW 함수의 경우 동일 순서면 앞 행의 함수 결과 값을 따르는데, CUME_DIST의 경우는 동일 순서면 뒤 행의 함수 결과값을 기준으로 한다.

라. NTILE 함수

NTILE 함수를 이용해 파티션별 전체 건수를 ARGUMENT 값으로 N 등분한 결과를 구할 수 있다.

[예제] 전체 사원을 급여가 높은 순서로 정렬하고, 급여를 기준으로 4개의 그룹으로 분류한다.

```
[예제] SELECT ENAME, SAL, NTILE(4) OVER (ORDER BY SAL DESC) as QUAR_TILE FROM EMP
```

```
[실행 결과] DEPTNO ENAME SAL QUAR_TILE ------ ------- ---- -------- 10 KING 5000 1 10 FORD 3000 1 10 SCOTT 3000 1 20 JONES 2975 1 20 BLAKE 2850 2 20 CL ARK 2450 2 20 ALLEN 1600 2 20 TURNER 1500 2 30 MILLER 1300 3 30 WARD 1250 3 30 MARTIN 1250 3 30 ADAMS 1100 4 30 JAMES 950 4 30 SMITH 800 4 1 4개의 행이 선택되었다.
```

위 예제에서 NTILE(4)의 의미는 14명의 팀원을 4개 조로 나눈다는 의미이다. 전체 14명을 4개의 집합으로 나누면 몫이 3명, 나머지가 2명이 된다. 나머지 두 명은 앞의 조부터 할당한다. 즉, 4명 + 4명 + 3명 + 3명으로 조를 나누게 된다.

1. DCL 개요

지금까지 살펴본 SQL 문장을 분류하면 테이블 생성과 조작에 관련된 명령어(DDL)와, 데이터를 조작하기 위한 명령어(DML), 그리고 TRANSACTION을 제어하기 위한 명령어(TCL)이다. 추가로, 이런 명령어들 이외에도 유저를 생성하고 권한을 제어할 수 있는 DCL(DATA CONTROL LANGUAGE) 명령어가 있다.

2. 유저와 권한

다른 부서 간에 또는 다른 회사 간에 데이터를 공유하기 위해 데이터베이스를 오픈해야 하는 경우가 가끔 발생한다. 물론 데이터베이스를 오픈하는 것 자체가 문제가 될 수 있다. 즉, 운영 시스템에서 사용하던 유저를 오픈하면 데이터의 손실 우려가 너무 커지게 되는 것이다. 이런 경우에 새로운 유저를 생성하고, 생성한 유저에게 공유할 테이블이나 기타 오브젝트에 대한 접근 권한만을 부여한다면 문제는 쉽게 해결할 수 있다. 일반적으로 회원제 웹사이트를 방문하여 서비스를 이용하려면 먼저 회원 가입을 해야 한다. 유저 아이디, 패스워드, 기타 개인정보를 입력하고 약관에 동의하면 회원 가입이 된다. 그리고 유저 아이디와 패스워드로 로그인하면 웹사이트의 서비스를 이용할 수 있게 된다. 그러나 영화나 유료 게임과 같은 특정 컨텐츠를 이용하려면 '권한이 없다'라는 메시지를 볼 수 있다. 여기서 유저 아이디와 패스워드를 유저라 할 수 있고, 유료 서비스에 대한 결재 여부를 권한이라 할 수 있다. 대부분의 데이터베이스는 데이터 보호와 보안을 위해서 유저와 권한을 관리하고 있는데, 예를 들어 Oracle을 설치하면 기본적으로 제공되는 유저들인 SYS, SYSTEM, SCOTT 유저에 대해서 [표 Ⅱ-2-8]을 통해서 간단하게 알아본다.

[표 Ⅱ-2-8] Oracle에서 제공하는 유저들

유저	역할
SCOTT	Oracle 테스트용 샘플 유저 Default 패스워드 : TIGER
SYS	DBA ROLE을 부여받은 유저
SYSTEM	데이터베이스의 모든 시스템 권한을 부여받은 DBA 유저 Oracle 설치 완료 시에 패스워드 설정

Oracle과 SQL Server의 사용자에 대한 아키텍처는 다른 면이 많다. Oracle은 유저를 통해 데이터베이스에 접속을 하는 형태이다. 즉, 아이디와 비밀번호 방식으로 인스턴스에 접속을 하고 그에 해당하는 스키마에 오브젝트 생성 등의 권한을 부여받게 된다. SQL Server는 인스턴스에 접속하기 위해 로그인이라는 것을 생성하게 되며, 인스턴스 내에 존재하는 다수의 데이터베이스에 연결하여 작업하기 위해 유저를 생성한 후 로그인과 유저를 매핑해 주어야 한다. 더 나아가 특정 유저는 특정 데이터베이스 내의 특정 스키마에 대해 권한을 부여받을 수 있다. SQL Server 로그인은 두 가지 방식으로 가능하다. 첫 번째, Windows 인증 방식으로 Windows에 로그인한 정보를 가지고 SQL Server에 접속하는 방식이다. Microsoft Windows 사용자 계정을 통해 연결되면 SQL Server는 운영 체제의 Windows 보안 주체 토큰을 사용하여 계정 이름과 암호가 유효한지 확인한다. 즉, Windows에서 사용자 ID를 확인한다. SQL Server는 암호를 요청하지 않으며 ID의 유효성 검사를 수행하지 않는다. Windows 인증은 기본 인증 모드이며 SQL Server 인증보다 훨씬 더 안전하다. Windows 인증은 Kerberos 보안 프로토콜을 사용하고, 암호 정책을 적용하여 강력한 암호에 대해 적합한 복잡성 수준을 유지하도록 하며, 계정 잠금 및 암호 만료를 지원한다. SQL Server가 Windows에서 제공하는 자격 증명을 신뢰하므로 Windows 인증을 사용한 연결을 트러스트된 연결이라고도 한다. 두 번째, 혼합 모드(Windows 인증 또는 SQL 인증) 방식으로 기본적으로 Windows 인증으로도 SQL Server에 접속 가능하며, Oracle의 인증과 같은 방식으로 사용자 아이디와 비밀번호로 SQL Server에 접속하는 방식이다. SQL 인증을 사용할 때는 강력한 암호(숫자+문자+특수문자 등을 혼합하여 사용)를 사용해야 한다. 예를 들어, 아래 [그림 Ⅱ-1-16]을 보면 SCOTT이라는 LOGIN 이름으로 인스턴스 INST1에 접속을 하여 미리 매핑되어 있는 SCOTT이라는 유저를 통해 PRODUCT라는 스키마에 속해 있는 ITEM이라는 테이블의 데이터를 액세스하고 있다.

[그림 Ⅱ-2-16] SQL Server 접속 경로

가. 유저 생성과 시스템 권한 부여

유저를 생성하고 데이터베이스에 접속한다. 하지만 데이터베이스에 접속했다고 해서 테이블, 뷰, 인덱스 등과 같은 오브젝트(OBJECT)를 생성할 수는 없다. 사용자가 실행하는 모든 DDL 문장(CREATE, ALTER, DROP, RENAME 등)은 그에 해당하는 적절한 권한이 있어야만 문장을 실행할 수 있다. 이러한 권한을 시스템 권한이라고 하며 약 100개 이상의 종류가 있다. 일반적으로 시스템 권한은 일일이 유저에게 부여되지 않는다. 100개 이상의 시스템 권한을 일일이 사용자에게 설정하는 것은 너무 복잡하고, 특히 유저로부터 권한을 관리하기가 어렵기 때문이다. 그래서 롤(ROLE)을 이용하여 간편하고 쉽게 권한을 부여하게 된다. 롤에 대한 자세한 설명은 차후에 하도록 하고 먼저 유저를 생성하고 권한을 부여한다. 새로운 유저를 생성하려면 일단 유저 생성 권한(CREATE USER)이 있어야 한다.

[예제] SCOTT 유저로 접속한 다음 PJS 유저(패스워드: KOREA7)를 생성해 본다.

[예제] Oracle CONN SCOTT/TIGER 연결되었다. CREATE USER PJS IDENTIFIED BY KOREA7; CREATE USER PJS IDENTIFIED BY KOREA7; * 1행에 오류: ERROR: 권한이 불충분하다

현재 SCOTT 유저는 유저를 생성할 권한을 부여받지 못했기 때문에 권한이 불충분하다는 오류가 발생한다. Oracle의 DBA 권한을 가지고 있는 SYSTEM 유저로 접속하면 유저 생성 권한(CREATE USER)을 다른 유저에게 부여할 수 있다.

[예제] SCOTT 유저에게 유저생성 권한(CREATE USER)을 부여한 후 다시 PJS 유저를 생성한다.

[예제 및 실행 결과] Oracle GRANT CREATE USER TO SCOTT; 권한이 부여되었다. CONN SCOTT/TIGER 연결되었다. CREATE USER PJS IDENTIFIED BY KOREA7; 사용자가 생성되었다.

SQL Server는 유저를 생성하기 전 먼저 로그인을 생성해야 한다. 로그인을 생성할 수 있는 권한을 가진 로그인은 기본적으로 sa이다.

[예제] sa로 로그인을 한 후 SQL 인증을 사용하는 PJS라는 로그인(패스워드: KOREA7)을 생성해 본다. 로그인 후 최초로 접속할 데이터베이스는 Adventure Works 데이터베이스로 설정한다.

[예제 및 실행 결과] SQL Server CREATE LOGIN PJS WITH PASSWORD='KOREA7', DEFAULT_DATABASE=AdventureWorks

SQL Server에서의 유저는 데이터베이스마다 존재한다. 그러므로 유저를 생성하기 위해서는 생성하고자 하는 유저가 속할 데이터베이스로 이동을 한 후 처리해야 한다.

[예제 및 실행 결과] SQL Server USE ADVENTUREWORKS; GO CREATE USER PJS FOR LOGIN PJS WITH DEFAULT_SCHEMA = dbo;

[예제] 생성된 PJS 유저로 로그인한다.

[예제 및 실행 결과] Oracle CONN PJS/KOREA7; 오류: ERROR: 사용자 PJS는 CREATE SESSION 권한을 가지고 있지 않음; 로그온이 거절되었다.

PJS 유저가 생성됐지만 아무런 권한도 부여받지 못했기 때문에 로그인을 하면 CREATE SESSION 권한이 없다는 오류가 발생한다. 유저가 로그인을 하려면 CREATE SESSION 권한을 부여받아야 한다.

[예제] PJS 유저가 로그인할 수 있도록 CREATE SESSION 권한을 부여한다.

[예제 및 실행 결과] Oracle CONN SCOTT/TIGER 연결되었다. GRANT CREATE SESSION TO PJS; 권한이 부여되었다. CONN PJS/KOREA7 연결되었다.

[예제] PJS 유저로 테이블을 생성한다.

[예제 및 실행 결과] Oracle SELECT * FROM TAB; 선택된 레코드가 없다. CREATE TABLE MENU (MENU_SEQ NUMBER NOT NULL, TITLE VARCHAR2(10)); CREATE TABLE MENU (* 1행에 오류: ERROR: 권한이 불충분하다.

[예제 및 실행 결과] SQL Server CREATE TABLE MENU (MENU_SEQ INT NOT NULL, TITLE VARCHAR(10)); 데이터베이스 'AdventureWorks'에서 CREATE TABLE 사용 권한이 거부되었다.

PJS 유저는 로그인 권한만 부여되었기 때문에 테이블을 생성하려면 테이블 생성 권한(CREATE TABLE)이 불충분하다는 오류가 발생한다.(Oracle, SQL Server)

[예제] SYSTEM 유저를 통하여 PJS 유저에게 CREATE TABLE 권한을 부여한 후 다시 테이블을 생성한다.

[예제 및 실행 결과] Oracle CONN SYSTEM/MANAGER 연결되었다. GRANT CREATE TABLE TO PJS; 권한이 부여되었다. CONN PJS/KOREA7 연결되었다. CREATE TABLE MENU (MENU_SEQ NUMBER NOT NULL, TITLE VARCHAR2(10)); 테이블이 생성되었다.

[예제 및 실행 결과] SQL Server GRANT CREATE TABLE TO PJS; 권한이 부여되었다. 스키마에 권한을 부여한다. GRANT Control ON SCHEMA::dbo TO PJS 권한이 부여되었다. PJS로 로그인한다. CREATE TABLE MENU (MENU_SEQ INT NOT NULL, TITLE VARCHAR(10)); 테이블이 생성되었다.

나. OBJECT에 대한 권한 부여

앞에서 PJS 유저를 생성하여 로그인하고 테이블을 만드는 과정에서 몇 가지의 권한에 대해서 살펴보았다. 이제는 특정 유저가 소유한 객체(OBJECT) 권한에 대해 알아본다. 오브젝트 권한은 특정 오브젝트인 테이블, 뷰 등에 대한 SELECT, INSERT, DELETE, UPDATE 작업 명령어를 의미한다. [표 Ⅱ-2-9]는 오브젝트 권한과 오브젝트와의 관계를 보여 주고 있다.

[표 Ⅱ-2-9] 오브젝트 권한과 오브젝트와의 관계 (Oracle 사례)

객체 권한	테이블	VIEWS	SEQUENCE	PROCEDURE
ALTER	○		○	
DELETE	○	○		
EXECUTE				○
INDEX	○			
INSERT	○	○		
REFERENCES	○			
SELECT	○	○	○	
UPDATE	○	○		

[표 Ⅱ-2-10] 오브젝트 권한과 오브젝트와의 관계 (SQL Server 사례)

객체 권한	테이블	VIEWS	FUNCTION	PROCEDURE
ALTER	○		○	
DELETE	○	○	○	
EXECUTE				○
INDEX	○			
INSERT	○	○		
REFERENCES	○			
SELECT	○	○	○	
UPDATE	○	○		

앞에서 PJS 유저가 생성한 MENU 테이블을 SCOTT 유저를 통해서 조회하면 어떻게 될까? SCOTT, PJS 뿐만 아니라 모든 유저는 각각 자신이 생성한 테이블 외에 다른 유저의 테이블에 접근하려면 해당 테이블에 대한 오브젝트 권한을 소유자로부터 부여받아야 한다. 우리가 남의 집에 방문했을 때 집주인의 허락 없이는 집에 들어갈 수 없는 것과 같은 이치이다. SQL Server도 같은 방식으로 동작한다. 한 가지 다른 점은 위에서 언급했듯이 유저는 단지 스키마에 대한 권한만을 가진다. 다시 말하면 테이블과 같은 오브젝트는 유저가 소유하는 것이 아니고 스키마가 소유를 하게 되며 유저는 스키마에 대해 특정한 권한을 가지는 것이다. 먼저 SCOTT 유저로 접속하여 PJS.MENU 테이블을 조회한다. 다른 유저가 소유한 객체에 접근하기 위해서는 객체 앞에 객체를 소유한 유저의 이름을 붙여서 접근해야 한다. SQL Server는 객체 앞에 소유한 유저의 이름을 붙이는 것이 아니고 객체가 속한 스키마 이름을 붙여야 한다.

[예제 및 실행 결과] Oracle CONN SCOTT/TIGER 연결되었다. SELECT * FROM PJS.MENU; SELECT * FROM PJS.MENU * 1행에 오류: ERROR: 테이블 또는 뷰가 존재하지 않는다.

[예제 및 실행 결과] SQL Server SCOTT로 로그인한다. SELECT * FROM dbo.MENU; 개체이름 'dbo.MENU'이(가) 잘못되었다.

SCOTT 유저는 PJS 유저로부터 MENU 테이블을 SELECT할 수 있는 권한을 부여받지 못했기 때문에 MENU 테이블을 조회할 수 없다.

[예제] PJS 유저로 접속하여 SCOTT 유저에게 MENU 테이블을 SELECT 할 수 있는 권한을 부여한다.

[예제 및 실행 결과] Oracle CONN PJS/KOREA7 연결되었다. INSERT INTO MENU VALUES (1, '화이팅'); 1개의 행이 만들어졌다. COMMIT; 커밋이 완료되었다. GRANT SELECT ON MENU TO SCOTT; 권한이 부여되었다.

[예제 및 실행 결과] SQL Server PJS로 로그인한다. INSERT INTO MENU VALUES (1, '화이팅'); 1개의 행이 만들어졌다. GRANT SELECT ON MENU TO SCOTT; 권한이 부여되었다.

다시 한 번 SCOTT 유저로 접속하여 PJS.MENU 테이블을 조회한다. 이제 PJS.MENU 테이블을 SELECT하면 테이블 자료를 볼 수 있다. SCOTT 유저는 PJS.MENU 테이블을 SELECT하는 권한만 부여 받았기 때문에 UPDATE, INSERT, DELETE와 같은 다른 작업을 할 수 없다. 오브젝트 권한은 SELECT, INSERT, DELETE, UPDATE 등의 권한을 따로따로 관리한다.

[예제] PJS.MENU 테이블에 UPDATE를 시도한다.

[예제 및 실행 결과] Oracle CONN SCOTT/TIGER 연결되었다. SELECT * FROM PJS.MENU; MENU_SEQ TITLE -------- -------- 1 화이팅 UPDATE PJS.MENU SET TITLE = '코리아' WHERE MENU_SEQ = 1; UPDATE PJS.MENU * 1행에 오=text>

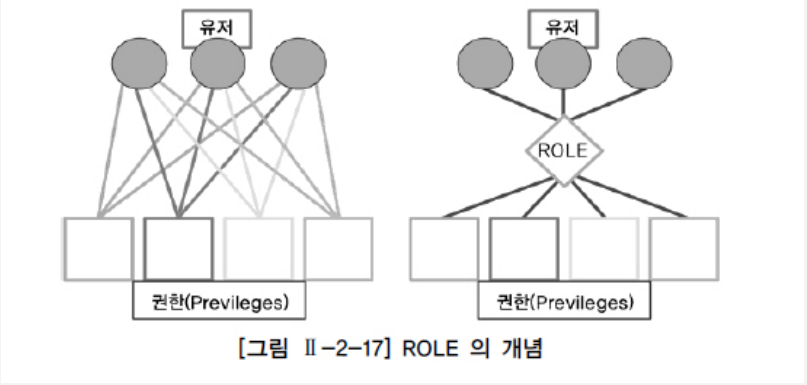
[예제 및 실행 결과] SQL Server SCOTT으로 로그인한다. SELECT * FROM PJS.MENU; MENU_SEQ TITLE -------- ---------- 1 화이팅 UPDATE PJS.MENU SET TITLE = '코리아' WHERE MENU_SEQ = 1; 개체 'MENU', 데이터베이스 'AdventureWorks', 스키마 'dbo'에 대한 UPDATE 권한이 거부되었다.

권한이 부족하여 UPDATE를 할 수 없다는 오류가 나타난다. PJS 유저에게 UPDATE 권한을 부여한 후 다시 시도하면 업데이트가 가능하다.

3. Role을 이용한 권한 부여

유저를 생성하면 기본적으로 CREATE SESSION, CREATE TABLE, CREATE PROCEDURE 등 많은 권한을 부여해야 한다. 데이터베이스 관리자는 유저가 생성될 때마다 각각의 권한들을 유저에게 부여하는 작업을 수행해야 하며 간혹 권한을 빠뜨릴 수도 있으므로 각 유저별로 어떤 권한이 부여되었는지를 관리해야 한다. 하지만 관리해야 할 유저가 점점 늘어나고 자주 변경되는 상황에서는 매우 번거로운 작업이 될 것이다. 이와 같은 문제를 줄이기 위하여 많은 데이터베이스에서 유저들과 권한들 사이에서 중개 역할을 하는 ROLE을 제공한다. 데이터베이스 관리자는 ROLE을 생성하고, ROLE에 각종 권한들을 부여한 후 ROLE을 다른 ROLE이나 유저에게 부여할 수 있다. 또한 ROLE에 포함되어 있는 권한들이 필요한 유저에게는 해당 ROLE만을 부여함으로써 빠르고 정확하게 필요한 권한을 부여할 수 있게 된다. [그림 Ⅱ-2-17]에서는 유저들과 권한들 사이 간 ROLE의 역할을 보여 주고

있다. 왼쪽 그림은 권한을 직접 유저에게 할당할 때를 나타내는 것이며, 오른쪽 그림은 ROLE에 권한을 부여한 후 ROLE을 유저들에게 부여하는 것을 나타내고 있다.



[그림 Ⅱ-2-17] ROLE 의 개념

ROLE에는 시스템 권한과 오브젝트 권한을 모두 부여할 수 있으며, ROLE은 유저에게 직접 부여될 수도 있고, 다른 ROLE에 포함하여 유저에게 부여될 수도 있다.

[예제] JISUNG 유저에게 CREATE SESSION과 CREATE TABLE 권한을 가진 ROLE을 생성한 후 ROLE을 이용하여 다시 권한을 할당한다. 권한을 취소할 때는 REVOKE를 사용한다.

[예제 및 실행 결과] Oracle CONN SYSTEM/MANAGER 연결되었다. REVOKE CREATE SESSION, CREATE TABLE FROM JISUNG; 권한이 취소되었다. CONN JISUNG/KOREA7 ERROR: 사용자 JISUNG은 CREATE SESSION 권한을 가지고 있지 않음. 로그온이 거절되었다.

[예제 및 실행 결과] SQL Server sa로 로그인한다. REVOKE CREATE TABLE FROM PJS; 권한이 취소되었다. PJS로 로그인한다. CREATE TABLE MENU (MENU_SEQ INT NOT NULL, TITLE VARCHAR(10)); 데이터베이스 'AdventureWorks'에서 CREATE TABLE사용 권한이 거부되었다.

[예제] 이제 LOGIN_TABLE이라는 ROLE을 만들고, 이 ROLE을 이용하여 JISUNG 유저에게 권한을 부여한다.

[예제 및 실행 결과] Oracle CONN SYSTEM/MANAGER 연결되었다. CREATE ROLE LOGIN_TABLE; 롤이 생성되었다. GRANT CREATE SESSION, CREATE TABLE TO LOGIN_TABLE; 권한이 부여되었다. GRANT LOGIN_TABLE TO JISUNG; 권한이 부여되었다. CONN JISUNG/KOREA7 연결되었다. CREATE TABLE MENU2(MENU_SEQ NUMBER NOT NULL, TITLE VARCHAR2(10)); 테이블이 생성되었다.

이와 같이 ROLE을 만들어 사용하는 것이 권한을 직접 부여하는 것보다 빠르고 안전하게 유저를 관리할 수 있는 방법이다. Oracle에서는 기본적으로 몇 가지 ROLE을 제공하고 있다. 그 중 가장 많이 사용하는 ROLE은 CONNECT와 RESOURCE이다. 참조를 위해 [표 Ⅱ-2-11]은 CONNECT와 RESOURCE ROLE에 부여된 권한 목록을 정리한 것이다. CONNECT는 CREATE SESSION과 같은 로그인 권한이 포함되어 있고, RESOURCE는 CREATE TABLE과 같은 오브젝트의 생성 권한이 포함되어 있다. 일반적으로 유저를 생성할 때 CONNECT와 RESOURCE ROLE을 사용하여 기본 권한을 부여한다.

[표 Ⅱ-2-11] CONNECT ROLE과 RESOURCE ROLE에 포함된 권한 목록 (Oracle 사례)

CONNECT	RESOURCE
ALTER SESSION	CREATE CLUSTER
CREATE CLUSTER	CREATE INDEXTYPE
CREATE DATABASE LINK	CREATE OPERATOR
CREATE MENU_SEQUENCE	CREATE PROCEDURE
CREATE SESSION	CREATE MENU_SEQUENCE
CREATE SYNONYM	CREATE TABLE
CREATE TABLE	CREATE TRIGGER
CREATE VIEW	CREATE

유저를 삭제하는 명령어는 DROP USER이고, CASCADE 옵션을 주면 해당 유저가 생성한 오브젝트를 먼저 삭제한 후 유저를 삭제한다.

[예제] 앞에서 MENU라는 테이블을 생성했기 때문에 CASCADE 옵션을 사용하여 JISUNG 유저를 삭제한 후, 유저 재생성 및 기본적인 ROLE을 부여한다.

[예제 및 실행 결과] Oracle CONN SYSTEM/MANAGER 연결되었다. DROP USER JISUNG CASCADE; 사용자가 삭제되었다. ☞ JISUNG 유저가 만든 MENU 테이블도 같이 삭제되었다. CREATE USER JISUNG IDENTIFIED BY KOREA7; 사용자가 생성되었다. GRANT CONNECT, RESOURCE TO JISUNG; 권한이 부여되었다. CONN JISUNG/KOREA7 연결되었다. CREATE TABLE MENU (MENU_SEQ NUMBER NOT NULL, TITLE VARCHAR2(10)); 테이블이 생성되었다.

SQL Server에서는 위와 같이 ROLE을 생성하여 사용하기보다는 기본적으로 제공되는 ROLE에 멤버로 참여하는 방식으로 사용한다. 특정 로그인이 멤버로 참여할 수 있는 서버 수준 역할(ROLE)은 [표 Ⅱ-2-12]와 같다.

[표 Ⅱ-2-12] 서버 수준 역할명 (SQL Server 사례)

서버 수준 역할명	설명
public	모든 SQL Server 로그인은 PUBLIC 서버 역할에 속한다. 서버 보안 주체에게 보안 객체에 대한 특정 사용 권한이 부여되지 않았거나 거부된 경우 사용자는 해당 개체에 대해 PUBLIC 으로 부여된 사용 권한을 상속 받는다. 모든 사용자가 개체를 사용할 수 있도록 하려는 경우에만 개체에 PUBLIC 권한을 할당해야 한다.
bulkadmin	BULK INSERT문을 수행할 수 있다.
dbcreator	데이터베이스를 생성, 변경, 삭제 및 복원할 수 있다.
diskadmin	디스크 파일을 관리하는데 사용된다.
processadmin	SQL Server의 인스턴스에서 실행중인 프로세스를 종료할 수 있다.
securityadmin	로그인 및 해당 속성을 관리한다. 서버 및 데이터베이스 수준의 사용 권한을 부여(GRANT), 거부(DENY), 취소(REVOKE)할 수 있다. 또한, 로그인의 암호를 다시 설정할 수 있다.
serveradmin	서버 차원의 구성 옵션을 변경하고 서버를 종료할 수 있다.
setupadmin	연결된 서버를 추가하거나 제거할 수 있다.
sysadmin	서버에서 모든 작업을 수행할 수 있다. 기본적으로 Windows BUILTIN ₩Administrators 그룹의 멤버인 로컬 관리자 그룹은 sysamdin 고정 서버 역할의 멤버이다.

데이터베이스에 존재하는 유저에 대해서는 아래와 같은 데이터베이스 역할의 멤버로 참여할 수 있다.

[표 Ⅱ-2-13] 데이터베이스 수준 역할명 (SQL Server 사례)

데이터베이스 수준 역할명	설명
db_accessadmin	Windows 로그인, Windows 그룹 및 SQL Server 로그인의 데이터베이스에 대한 액세스를 추가하거나 제거할 수 있다.
db_backupoperator	데이터베이스를 백업할 수 있다.
db_datareader	모든 사용자 테이블의 모든 데이터를 읽을 수 있다.
db_datawriter	모든 사용자 테이블에서 데이터를 추가, 삭제, 변경할 수 있다.
db_ddladmin	데이터베이스에서 모든 DDL 명령을 수행할 수 있다.
db_denydatareader	데이터베이스 내에 있는 사용자 테이블의 데이터를 읽을 수 없다.
db_denydatawriter	데이터베이스 내의 모든 사용자 테이블에 있는 데이터를 추가, 삭제, 변경할 수 없다.
db_owner	데이터베이스 내에 있는 모든 구성 및 유지관리 작업을 수행할 수 있고 데이터베이스를 삭제할 수도 있다.
db_securityadmin	역할 멤버 자격을 수정하고 사용 권한 관리를 할 수 있다. 이 역할에 보안 주체를 추가하면 원하지 않는 권한 상승이 설정될 수 있다.

SQL Server에서는 Oracle과 같이 Role을 자주 사용하지 않는다. 대신 위에서 언급한 서버 수준 역할 및 데이터베이스 수준 역할을 이용하여 로그인 및 사용자 권한을 제어한다. 인스턴스 수준의 작업이 필요한 경우 서버 수준 역할을 부여하고 그보다 작은 개념인 데이터베이스 수준의 권한이 필요한 경우 데이터베이스 수준의 역할을 부여하면 된다. 즉, 인스턴스 수준을 요구하는 로그인에는 서버 수준 역할을, 데이터베이스 수준을 요구하는 사용자에게는 데이터베이스 수준 역할을 부여한다.

1. 절차형 SQL 개요

일반적인 개발 언어처럼 SQL에도 절차 지향적인 프로그램이 가능하도록 DBMS 벤더별로 PL(Procedural Language)/SQL(Oracle), SQL/PL(DB2), T-SQL(SQL Server) 등의 절차형 SQL을 제공하고 있다. 절차형 SQL을 이용하면 SQL문의 연속적인 실행이나 조건에 따른 분기처리를 이용하여 특정 기능을 수행하는 저장 모듈을 생성할 수 있다. 본 절에서는 절차형 SQL을 이용하여 만들 수 있는 저장 모듈인 Procedure, User Defined Function, Trigger에 대해서 간단하게 살펴본다. (상세한 내역은 각 DBMS 벤더의 매뉴얼을 참조한다.)

2. PL/SQL 개요

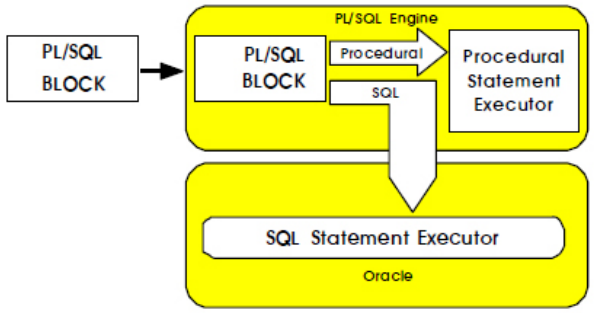
가. PL/SQL 특징

Oracle의 PL/SQL은 Block 구조로 되어있고 Block 내에는 DML 문장과 QUERY 문장, 그리고 절차형 언어(IF, LOOP) 등을 사용할 수 있으며, 절차적 프로그래밍을 가능하게 하는 트랜잭션 언어이다. 이런 PL/SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있다. 저장 모듈이란 PL/SQL 문장을 데이터베이스 서버에 저장하여 사용자와 애플리케이션 사이에서 공유할 수 있도록 만든 일종의 SQL 컴포넌트 프로그램이며, 독립적으로 실행되거나 다른 프로그램으로부터 실행될 수 있는 완전한 실행 프로그램이다. Oracle의 저장 모듈에는 Procedure, User Defined Function, Trigger가 있다.

PL/SQL의 특징은 다음과 같다.

- PL/SQL은 Block 구조로 되어있어 각 기능별로 모듈화가 가능하다. - 변수, 상수 등을 선언하여 SQL 문장 간 값을 교환한다. - IF, LOOP 등의 절차형 언어를 사용하여 절차적인 프로그램이 가능하도록 한다. - DBMS 정의 에러나 사용자 정의 에러를 정의하여 사용할 수 있다. - PL/SQL은 Oracle에 내장되어 있으므로 Oracle과 PL/SQL을 지원하는 어떤 서버로도 프로그램을 옮길 수 있다. - PL/SQL은 응용 프로그램의 성능을 향상시킨다. - PL/SQL은 여러 SQL 문장을 Block으로 묶고 한 번에 Block 전부를 서버로 보내기 때문에 통신량을 줄일 수 있다.

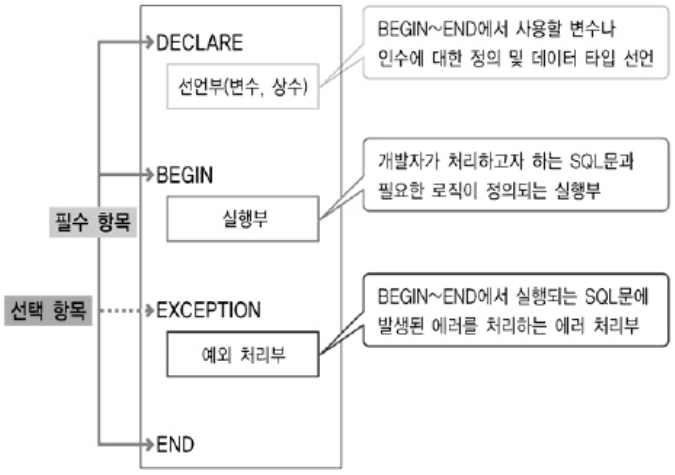
[그림 Ⅱ-2-18]은 PL/SQL Architecture이다. PL/SQL Block 프로그램을 입력받으면 SQL 문장과 프로그램 문장을 구분하여 처리한다. 즉 프로그램 문장은 PL/SQL 엔진이 처리하고 SQL 문장은 Oracle 서버의 SQL Statement Executor가 실행하도록 작업을 분리하여 처리한다.



[그림 Ⅱ-2-18] Oracle의 PL/SQL 엔진

나. PL/SQL 구조

다음은 PL/SQL의 블록 구조를 표현한 내용이다.



[그림 Ⅱ-2-19] PL/SQL 블록 구조

- DECLARE : BEGIN ~ END 절에서 사용될 변수와 인수에 대한 정의 및 데이터 타입을 선언하는 선언부이다. - BEGIN ~ END : 개발자가 처리하고자 하는 SQL문과 여러 가지 비교문, 제어문을 이용하여 필요한 로직을 처리하는 실행부이다. - EXCEPTION : BEGIN ~ END 절에서 실행되는 SQL문이 실행될 때 에러가 발생하면 그 에러를 어떻게 처리할 것이지를 정의하는 예외 처리부이다.

다. PL/SQL 기본 문법(Syntax)

앞으로 살펴볼 User Defined Function이나 Trigger의 생성 방법이나 사용 목적은 다르지만 기본적인 문법은 비슷하기 때문에 여기에서는 Stored Procedure를 통해서 PL/SQL에 대한 기본적인 문법을 정리한다.

```
CREATE [OR REPLACE] Procedure [Procedure_name] ( argument1 [mode] data_type1, argument2 [mode] date_type2, ... ... ) IS [AS] ... ... BEGIN ... ... EXCEPTION ... ... END; /
```

다음은 생성된 프로시저를 삭제하는 명령어이다.

```
DROP Procedure [Procedure_name];
```

CREATE TABLE 명령어로 테이블을 생성하듯 CREATE 명령어로 데이터베이스 내에 프로시저를 생성할 수 있다. 이렇게 생성한 프로시저는 데이터베이스 내에 저장된다. 프로시저는 개발자가 자주 실행해야 하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈이기 때문에 필요할 때 호출하여 실행할 수 있다. [OR REPLACE] 절은 데이터베이스 내에 같은 이름의 프로시저가 있을 경우, 기존의 프로시저를 무시하고 새로운 내용으로 덮어쓰기 하겠다는 의미이다. Argument는 프로시저가 호출될 때 프로시저 안으로 어떤 값이 들어오거나 혹은 프로시저에서 처리한 결과값을 운영 체제로 리턴시킬 매개 변수를 지정할 때 사용한다. [mode] 부분에 지정할 수 있는 매개 변수의 유형은 3가지가 있다. 먼저 IN은 운영 체제에서 프로시저로 전달될 변수의 MODE이고, OUT은 프로시저에서 처리된 결과가 운영체제로 전달되는 MODE이다. 마지막으로 잘 쓰지는 않지만 INOUT MODE가 있는데 이 MODE는 IN과 OUT 두 가지의 기능을 동시에 수행하는 MODE이다. 마지막에 있는 슬래쉬("/")는 데이터베이스에게 프로시저를 컴파일하라는 명령어이다. 앞에서 잠깐 언급했지만 PL/SQL과 관련된 내용은 상당히 다양하고 분량이 많기 때문에 본 가이드에서는 간단한 문법과 사용 목적에 초점을 맞춰 이해하기 바란다.

3. T-SQL 개요

가. T-SQL 특징

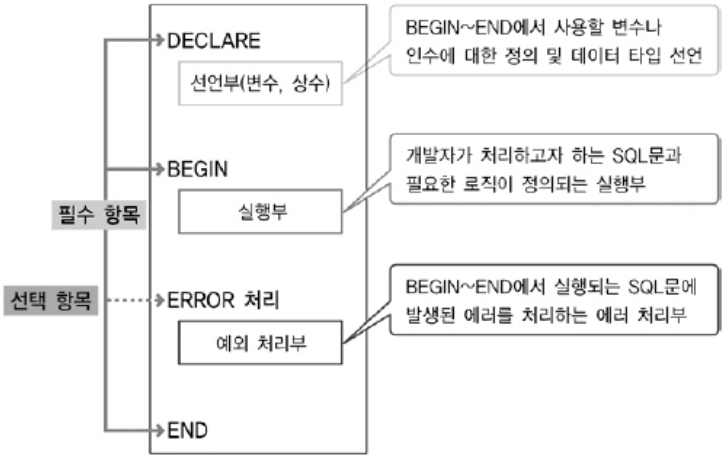
T-SQL은 근본적으로 SQL Server를 제어하기 위한 언어로서, T-SQL은 엄격히 말하면, MS사에서 ANSI/ISO 표준의 SQL에 약간의 기능을 더 추가해 보완적으로 만든 것이다. T-SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있는데, T-SQL의 프로그래밍 기능은 아래와 같다.

- 변수 선언 기능 @@이라는 전역변수(시스템 함수)와 @이라는 지역변수가 있다. - 지역변수는 사용자가 자신의 연결 시간 동안만 사용하기 위해 만들어지는 변수이며 전역변수는 이미 SQL서버에 내장된 값이다. - 데이터 유형(Data Type)을 제공한다. 즉 int, float, varchar 등의 자료형을 의미한다. - 연산자(Operator) 산술연산자(+, -, *, /)와 비교연산자(=, <, >, <>) 논리연산자(and, or, not) 사용이 가능하다. - 흐름 제어 기능 IF-ELSE와 WHILE, CASE-THEN 사용이 가능하다. - 주석 기능한줄 주석 : -- 뒤의 내용은 주석범위 주석 : /* 내용 */ 형태를 사용하며, 여러 줄도 가능함

T-SQL과 타 DBMS가 제공하는 SQL은 약간만 다를 뿐 그 맥락은 같이 하기 때문에, 조금의 변경 사항만 적용하면 같은 기능을 수행할 수 있다. 그리고 많은 사람들이 SQL 서버에 엔터프라이즈 매니저의 UI를 통하여 접근하는 경우가 많은데, 실??이 더 바람직하다.

나. T-SQL 구조

다음은 T-SQL의 구조를 표현한 내용이다. PL/SQL과 유사하다.



[그림 Ⅱ-2-20] T-SQL 구조

- DECLARE : BEGIN ~ END 절에서 사용될 변수와 인수에 대한 정의 및 데이터 타입을 선언하는 선언부이다. - BEGIN ~ END : 개발자가 처리하고자 하는 SQL문과 여러 가지 비교문, 제어문을 이용하여 필요한 로직을 처리하는 실행부이다. T-SQL에서는 BEGIN, END 문을 반드시 사용해야하는 것은 아니지만 블록 단위로 처리하고자 할 때는 반드시 작성해야 한다. - ERROR 처리 : BEGIN ~ END 절에서 실행되는 SQL문이 실행될 때 에러가 발생하면 그 에러를 어떻게 처리할 것이지를 정의하는 예외 처리부이다.

다. T-SQL 기본 문법(Syntax)

앞으로 살펴볼 User Defined Function이나 Trigger의 생성 방법과 사용 목적은 Stored Procedure와 다르지만 기본적인 문법은 비슷하기 때문에 여기에서는 Stored Procedure를 통해서 T-SQL에 대한 기본적인 문법을 정리한다.

```
CREATE Procedure [schema_name.]Procedure_name @parameter1 data_type1 [mode], @parameter2 date_type2 [mode], ... ... WITH AS ... ... BEGIN ... ... ERROR 처리 ... ... END;
```

다음은 생성된 프로시저를 삭제하는 명령어이다.

```
DROP Procedure [schema_name.]Procedure_name;
```

CREATE TABLE 명령어로 테이블을 생성하듯 CREATE 명령어로 데이터베이스 내에 프로시저를 생성할 수 있다. 이렇게 생성한 프로시저는 데이터베이스 내에 저장된다. 프로시저는 개발자가 자주 실행해야 하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈이기 때문에 필요할 때 호출하여 실행할 수 있다. 프로시저의 변경이 필요할 경우 Oracle은 [CREATE OR REPLACE]와 같이 하나의 구문으로 처리하지만 SQL Server는 CREATE 구문을 ALTER 구문으로 변경하여야 한다. @parameter는 프로시저가 호출될 때 프로시저 안으로 어떤 값이 들어오거나 혹은 프로시저에서 처리한 결과 값을 리턴 시킬 매개 변수를 지정할 때 사용한다. [mode] 부분에 지정할 수 있는 매개 변수(@parameter)의 유형은 4가지가 있다.

① VARYING결과 집합이 출력 매개 변수로 사용되도록 지정합니다. CURSOR 매개변수에만 적용된다. ② DEFAULT지정된 매개변수가 프로시저를 호출할 당시 지정되지 않을 경우 지정된 기본값으로 처리한다. 즉, 기본 값이 지정되어 있으면 해당 매개 변수를 지정하지 않아도 프로시저가 지정된 기본 값으로 정상적으로 수행이 된다. ③ OUT, OUTPUT프로시저에서 처리된 결과 값을 EXECUTE 문 호출 시 반환한다. ④ READONLY자주 사용되지는 않는다. 프로시저 본문 내에서 매개 변수를 업데이트하거나 수정할 수 없음을 나타낸다. 매개 변수 유형이 사용자 정의 테이블 형식인 경우 READONLY를 지정해야 한다.

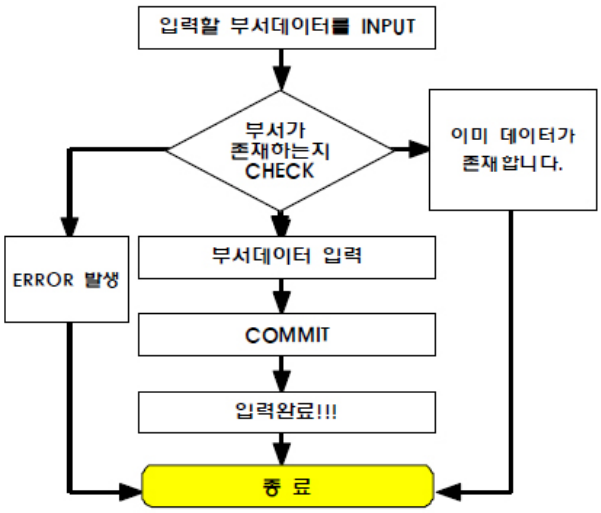
WITH 부분에 지정할 수 있는 옵션은 3가지가 있다.

① RECOMPILE데이터베이스 엔진에서 현재 프로시저의 계획을 캐시하지 않고 프로시저가 런타임에 컴파일 된다. 데이터베이스 엔진에서 저장 프로시저 안에 있는 개별 쿼리에 대한 계획을 삭제하려 할 때 RECOMPILE 쿼리 힌트를 사용한다. ② ENCRYPTIONCREATE PROCEDURE 문의 원본 텍스트가 알아보기 어려운 형식으로 변환된다. 변조된 출력은 SQL Server의 카탈로그 뷰 어디에서도 직접 표시되지 않는다. 원본을 볼 수 있는 방법이 없기 때문에 반드시 원본은 백업을 해두어야 한다. ③ EXECUTE AS 해당 저장 프로시저를 실행할 보안 컨텍스트를 지정한다.

앞에서 잠깐 언급했지만 T-SQL과 관련된 내용은 상당히 다양하고 분량이 많기 때문에 본 가이드에서는 간단한 문법과 사용 목적에 초점을 맞춰 이해하기 바란다.

4. Procedure의 생성과 활용

[그림 Ⅱ-2-21]은 앞으로 생성할 Procedure의 기능을 Flow Chart로 나타낸 그림이다.



[그림 Ⅱ-2-21] 부서 입력 FLOW CHART

[예제] SCOTT 유저가 소유하고 있는 DEPT 테이블에 새로운 부서를 등록하는 Procedure를 작성한다. SCOTT 유저가 기본적으로 소유한 DEPT 테이블의 구조는 [표 Ⅱ-2-14]와 같다.

[표 Ⅱ-2-14] DEPT 테이블 구조

DEPT			
칼럼ID	TYPE	길이	인덱스
DEPTNO	NUMBER	2	PK
DNAME	VARCHAR2	14	
LOC	VARCHAR2	13	

[예제] Oracle CREATE OR REPLACE Procedure p_DEPT_insert ----------① (v_DEPTNO in number, v_dname in varchar2, v_loc in varchar2, v_result out varchar2) IS cnt number := 0; BEGIN SELECT COUNT(*) INTO CNT ----------② FROM DEPT WHERE DEPTNO = v_DEPTNO AND ROWNUM = 1; if cnt > 0 then ----------③ v_result := '이미 등록된 부서번호이다'; else INSERT INTO DEPT (DEPTNO, DNAME, LOC) ----------④ VALUES (v_DEPTNO, v_dname, v_loc); COMMIT; ----------⑤ v_result := '입력 완료!!'; end if; EXCEPTION ----------⑥ WHEN OTHERS THEN ROLLBACK; v_result := 'ERROR 발생'; END; /

[예제] SQL Server CREATE Procedure dbo.p_DEPT_insert ----------① @v_DEPTNO int, @v_dname varchar(30), @v_loc varchar(30), @v_result varchar(100) OUT PUT AS DECLARE @cnt int SET @cnt = 0 BEGIN SELECT @cnt=COUNT(*) ----------② FROM DEPT WHERE DEPTNO = @v_DEPTNO IF @cnt > 0 ----------③ BE GIN SET @v_result = '이미 등록된 부서번호이다' RETURN END ELSE BEGIN BEGIN TRAN INSERT INTO DEPT (DEPTNO, DNAME, LOC) ----------④ VALUES (@v_DEPTNO, @v_dname, @v_loc) IF @@ERROR<>0 BEGIN ROLLBACK ----------⑥ SET @v_result = 'ERROR 발생' RETURN END ELSE BEGIN COMMIT ---------- ----------⑤ SET @v_result = '입력 완료!!' RETURN END END END

DEPT 테이블은 DEPTNO 칼럼이 PRIMARY KEY로 설정되어 있으므로, DEPTNO 칼럼에는 유일한 값을 넣어야만 한다. [예제]에 대한 설명은 다음과 같다.

① DEPT 테이블에 들어갈 칼럼 값(부서코드, 부서명, 위치)을 입력 받는다. ② 입력 받은 부서코드가 존재하는지 확인한다. ③ 부서코드가 존재하면 '이미 등록된 부서번호입니다'라는 메시지를 출력 값에 넣는다. ④ 부서코드가 존재하지 않으면 입력받은 필드 값으로 새로운 부서 레코드를 입력한다. ⑤ 새로운 부서가 정상적으로 입력됐을 경우에는 COMMIT 명령어를 통해서 트랜잭션을 종료한다. ⑥ 에러가 발생하면 모든 트랜잭션을 취소하고 'ERROR 발생'라는 메시지를 출력값에 넣는다.

앞에 있는 프로시저를 작성하면서 주의해야 할 몇 가지 문법적 요소가 있다. 첫째, PL/SQL 및 T-SQL에서는 다양한 변수가 있다. 예제에서 나온 cnt라는 변수를 SCALAR 변수라고 한다. SCALAR 변수는 사용자의 임시 데이터를 하나만 저장할 수 있는 변수이며 거의 모든 형태의 데이터 유형을 지정할 수 있다. 둘째, PL/SQL에서 사용하는 SQL 구문은 대부분 지금까지 살펴본 것과 동일하게 사용할 수 있지만 SELECT 문장은 다르다. PL/SQL에서 사용하는 SELECT 문장은 결과값이 반드시 있어야 하며, 그 결과 역시 반드시 하나여야 한다. 조회 결과가 없거나 하나 이상인 경우에는 에러를 발생시킨다. T-SQL에서는 결과 값이 없어도 에러가 발생하지 않는다. 셋째, T-SQL을 비롯하여 일반적으로 대입 연산자는 “=”을 사용하지만 PL/SQL에서는 “:=”를 사용한다. 넷째, 에러 처리를 담당하는 EXCEPTION에는 WHEN ~ THEN 절을 사용하여 에러의 종류별로 적절히 처리한다. OTHERS를 이용하여 모든 에러를 처리할 수 있지만 정확하게 에러를 처리하는 것이 좋다. T-SQL에서는 에러 처리를 다양하게 처리할 수 있으며 위의 예제는 그 한 예이다. 다음은 지금까지 작성한 프로시저를 실행하여 기능을 테스트한 과정이다.

[실행 결과] Oracle SQL> SELECT * FROM DEPT; ----------① DEPTNO DNAME LOC ------- ------- --------- 10 ACCOUNTING NEW YORK 20 RESEARCH DALLAS 3 0 SALES CHICAGO 40 OPERATIONS BOSTON SQL> variable rslt varchar2(30); ----------② SQL> EXECUTE p_DEPT_insert(10,'dev','seoul',:rslt); ----------③ PL/SQL 처리가 정상적으로 완료되었다. SQL> print rslt; ----------④ RSLT ------------------------------ 이미 등록된 부서번호이다 SQL> EXECUTE p_DEPT_insert(5 0,'NewDev','seoul',:rslt); ----------⑤ PL/SQL 처리가 정상적으로 완료되었다. SQL> print rslt; ----------⑥ RSLT ------------------------------ 입력 완료!! SQL> SELECT * FROM DEPT; ----------⑦ DEPTNO DNAME LOC ------ -------- --------- 10 ACCOUNTING NEW YORK 20 RESEARCH DALLAS 30 SALES CHICAGO 40 OPERATI ONS BOSTON 50 NewDev SEOUL 5개의 행이 선택되었다.

① DEPT 테이블을 조회하면 총 4개 행의 결과가 출력된다. ② Procedure를 실행한 결과 값을 받을 변수를 선언한다. (BIND 변수) ③ 존재하는 DEPTNO(1 0)를 가지고 Procedure를 실행한다. ④ DEPTNO가 10인 부서는 이미 존재하기 때문에 변수 rslt를 print해 보면 '이미 등록된 부서번호이다' 라고 출력된다. ⑤ 이번에는 새로운 DEPTNO(50)를 가지고 입력한다. ⑥ rslt를 출력해 보면 '입력 완료!!' 라고 출력된다. ⑦ DEPT 테이블을 조회하여 보면 DEPTNO가 50인 데이터가 정확하게 저장되었음을 확인할 수 있다.

T-SQL로 작성한 프로시저를 실행하기 위해서는 일반적으로 SQL Server에서 제공하는 기본 클라이언트 프로그램인 SQL Server MANAGEMENT STUDIO를 사용한다.

[실행 결과] SQL Server SELECT * FROM DEPT; ----------① DEPTNO DNAME LOC ------- ---------- --------- 10 ACCOUNTING NEW YORK 20 RESEARCH DALLAS 30 SALES CHICAGO 40 OPERATIONS BOSTON DECALRE @v_result VARCHAR(100) ----------② EXECUTE dbo.p_DEPT_insert 10, 'dev', 'seoul', @v_result=@v_result OUTPUT ----------③ SELECT @v_result AS RSLT ----------④ RSLT ---------- 이미 등록된 부서번호이다 DECALRE @v_result VARCHAR(100) ----------⑤ EXECUTE dbo.p_DEPT_insert 50, 'dev', 'seoul', @v_result=@v_result OUTPUT ----------⑥ SELECT @v_result AS RSLT ----------⑦ RSLT ---------- 입력 완료! SELECT * FROM DEPT; ----------⑧ DEPTNO DNAME LOC ------- -------- --------- 10 ACCOUNTING NEW YORK 20 RESEARCH DALLAS 30 SALES CHICAGO 40 OPERATIONS BOSTON 50 NewDev SEOUL 5개의 행에서 선택되었다.

① DEPT 테이블을 조회하면 총 4개 행의 결과가 출력된다. ② Procedure를 실행한 결과 값을 받을 변수를 선언한다. ③ 존재하는 DEPTNO(10)를 가지고 Procedure를 실행한다. ④ DEPTNO가 10인 부서는 이미 존재하기 때문에 변수 rslt를 print해 보면 '이미 등록된 부서번호이다'라고 출력된다. ⑤ Procedure를 실행한 결과 값을 받을 변수를 선언한다. ⑥ 이번에는 새로운 DEPTNO(50)를 가지고 입력한다. ⑦ rslt를 출력해 보면 '입력 완료!'라고 출력된다. ⑧ DEPT 테이블을 조회하여 보면 DEPTNO가 50인 데이터가 정확하게 저장되었음을 확인할 수 있다.

5. User Defined Function의 생성과 활용

User Defined Function은 Procedure처럼 절차형 SQL을 로직과 함께 데이터베이스 내에 저장해 놓은 명령문의 집합을 의미한다. 앞에서 학습한 SUM, SUBSTR, NVL 등의 함수는 벤더에서 미리 만들어둔 내장 함수이고, 사용자가 별도의 함수를 만들 수도 있다. Function이 Procedure와 다른 점은 RETURN을 사용해서 하나의 값을 반드시 되돌려 줘야 한다는 것이다. 즉 Function은 Procedure와는 달리 SQL 문장에서 특정 작업을

[예제] K-리그 8월 경기결과와 두 팀간의 점수차를 ABS 함수를 사용하여 절대값으로 출력한다.

[예제] Oracle SELECT SCHE_DATE 경기일자, HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들, HOME_SCORE || ' - ' || AWAY_SCORE SCORE, ABS(HOME_SCORE - AWAY_SCORE) 점수차 FROM SCHEDULE WHERE GUBUN = 'Y' AND SCHE_DATE BETWEEN '20120801' AND '20120831' ORDER BY SCHE_DATE;

[예제] SQL Server SELECT SCHE_DATE 경기일자, HOMETEAM_ID + ' - ' + AWAYTEAM_ID AS 팀들, HOME_SCORE + ' - ' + AWAY_SCORE AS SCORE, ABS(HOME_SCORE - AWAY_SCORE) AS 점수차 FROM SCHEDULE WHERE GUBUN = 'Y' AND SCHE_DATE BETWEEN '20120801' AND '20120831' ORDER BY SCHE_DATE;

[실행 결과] 경기일자 팀들 SCORE 점수차 ------- -------- ----- ----- 20120803 K01 - K03 3 - 0 3 20120803 K06 - K09 2 - 1 1 20120803 K08 - K07 1 - 0 1 20120804 K05 - K04 2 - 1 1 20120804 K10 - K02 0 - 3 3 20120811 K07 - K10 1 - 1 0 20120811 K03 - K08 2 - 0 2 20120811 K09 - K05 0 - 1 1 20120811 K04 - K02 0 - 2 2 20120811 K01 - K06 0 - 0 0 20120818 K05 - K01 0 - 2 2 20120818 K02 - K09 1 - 2 1 20120818 K08 - K10 3 - 1 2 20120818 K04 - K07 1 - 0 1 20120818 K06 - K03 3 - 1 2 20120824 K02 - K01 1 - 1 0 20120824 K05 - K03 3 - 3 0 20120824 K08 - K06 4 - 3 1 20120825 K10 - K04 1 - 1 0 20120825 K09 - K07 1 - 1 0 20120828 K04 - K08 2 - 3 1 20120828 K09 - K10 2 - 0 2 20120828 K03 - K02 0 - 0 0 20120828 K01 - K07 0 - 1 1 20120828 K06 - K05 1 - 1 0 25개의 행이 선택되었다.

[예제]에서 사용한 ABS 함수를 만드는데, INPUT 값으로 숫자만 들어온다고 가정한다.

[예제] Oracle CREATE OR REPLACE Function UTIL_ABS (v_input in number) ---------- ① return NUMBER IS v_return number := 0; ---------- ② BEGIN if v_input < 0 then ---------- ③ v_return := v_input * -1; else v_return := v_input; end if; RETURN v_return; ---------- ④ END; /

[예제] SQL Server CREATE Function dbo.UTIL_ABS (@v_input int) ---------- ① RETURNS int AS BEGIN DECLARE @v_return int ---------- ② SET @v_return=0 IF @v_input < 0 ---------- ③ SET @v_return = @v_input * -1 ELSE SET @v_return = @v_input RETURN @v_return; ---------- ④ END

[예제]에서 생성한 UTIL_ABS Function의 처리 과정은 다음과 같다.

① 숫자 값을 입력 받는다. 예제에서는 숫자 값만 입력된다고 가정한다. ② 리턴 값을 받아 줄 변수인 v_return를 선언한다. ③ 입력 값이 음수이면 -1을 곱하여 v_return 변수에 대입한다. ④ v_return 변수를 리턴한다.

[예제] 함수를 이용하여 앞의 SQL을 수정하여 실행한다.

[예제] Oracle SELECT SCHE_DATE 경기일자, HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들, HOME_SCORE || ' - ' || AWAY_SCORE SCORE, UTIL_ABS(HOME_SCORE - AWAY_SCORE) 점수차 FROM SCHEDULE WHERE GUBUN = 'Y' AND SCHE_DATE BETWEEN '20120801' AND '20120831' ORDER BY SCHE_DATE;

[예제] SQL Server SELECT SCHE_DATE 경기일자, HOMETEAM_ID + ' - ' + AWAYTEAM_ID AS 팀들, HOME_SCORE + ' - ' + AWAY_SCORE AS SCORE, dbo.UTIL_ABS(HOME_SCORE - AWAY_SCORE) AS 점수차 FROM SCHEDULE WHERE GUBUN = 'Y' AND SCHE_DATE BETWEEN '20120801' AND '20120831' ORDER BY SCHE_DATE;

[실행 결과] 경기일자 팀들 SCORE 점수차 ------- -------- ------ ------ 20120803 K01 - K03 3 - 0 3 20120803 K06 - K09 2 - 1 1 20120803 K08 - K07 1 - 0 1 20120804 K05 - K04 2 - 1 1 20120804 K10 - K02 0 - 3 3 25개의 행이 선택되었다.

실행 결과는 앞의 ABS 내장함수를 사용한 SQL 문장과 같은 결과를 확인할 수 있다.

6. Trigger의 생성과 활용

Trigger란 특정한 테이블에 INSERT, UPDATE, DELETE와 같은 DML문이 수행되었을 때, 데이터베이스에서 자동으로 동작하도록 작성된 프로그램이다. 즉 사용자가 직접 호출하여 사용하는 것이 아니고 데이터베이스에서 자동적으로 수행하게 된다. Trigger는 테이블과 뷰, 데이터베이스 작업을 대상으로 정의할 수 있으며, 전체 트랜잭션 작업에 대해 발생되는 Trigger와 각 행에 대해서 발생되는 Trigger가 있다. 요구 사항은 다음과 같다고 가정한다. 어떤 쇼핑몰에 하루에 수만 건의 주문이 들어온다. 주문 데이터는 주문일자, 주문상품, 수량, 가격이 있으며, 사장을 비롯한 모든 임직원이 일자별, 상품별 총 판매수량과 총 판매가격으로 구성된 주문 실적을 온라인상으로 실시간 조회한다고 했을 때, 한 사람의 임직원이 조회할 때마다 수만 건의 데이터를 읽어 계산해야 한다. 가끔 한 번씩 조회한다면 문제가 없을 수도 있으나 빈번하게 조회작업이 일어난다면 조회작업에 많은 시간을 허비할 수 있다.

[예제] 트리거(Trigger)를 사용하여 주문한 건이 입력될 때마다, 일자별 상품별로 판매수량과 판매금액을 집계하여 집계자료를 보관하도록 한다. 먼저 관련 테이블을 생성한다.

[표 Ⅱ-2-15] 주문정보 테이블

ORDER_LIST		
필드명	TYPE	길이
ORDER_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

[표 Ⅱ-2-16] 일자별 판매집계 테이블

SALES_PER_DATE		
필드명	TYPE	길이
SALE_DATE	CHAR	8
PRODUCT	VARCHAR2	10
QTY	NUMBER	
AMOUNT	NUMBER	

[예제] Oracle CREATE TABLE ORDER_LIST (ORDER_DATE CHAR(8) NOT NULL, PRODUCT VARCHAR2(10) NOT NULL, QTY NUMBER NOT NULL, AMOUNT NUMBER NOT NULL); CREATE TABLE SALES_PER_DATE (SALE_DATE CHAR(8) NOT NULL, PRODUCT VARCHAR2(10) NOT NULL, QTY NUMBER NOT NULL, AMOUNT NUMBER NOT NULL);

[예제] SQL Server CREATE TABLE ORDER_LIST (ORDER_DATE CHAR(8) NOT NULL, PRODUCT VARCHAR(10) NOT NULL, QTY INT NOT NULL, AMOUNT INT NOT NULL); CREATE TABLE SALES_PER_DATE (SALE_DATE CHAR(8) NOT NULL, PRODUCT VARCHAR(10) NOT NULL, QTY INT NOT NULL, AMOUNT INT NOT NULL);

[예제] 이제 Trigger를 작성한다. Trigger의 역할은 ORDER_LIST에 주문 정보가 입력되면 주문 정보의 주문 일자(ORDER_LIST.ORDER_DATE)와 주문 상품(ORDER_LIST.PRODUCT)을 기준으로 판매 집계 테이블(SALES_PER_DATE)에 해당 주문 일자의 주문 상품 레코드가 존재하면 판매 수량과 판매 금액을 더하고 존재하지 않으면 새로운 레코드를 입력한다.

[예제] Oracle CREATE OR REPLACE Trigger SUMMARY_SALES ---------- ① AFTER INSERT ON ORDER_LIST FOR EACH ROW DECLARE ---------- ② o_date ORDER_LIST.order_date%TYPE; o_prod ORDER_LIST.product%TYPE; BEGIN o_date := :NEW.order_date; o_prod := :NEW.product; UPDATE SALES_PER_DATE ---------- ③ SET qty = qty + :NEW.qty, amount = amount + :NEW.amount WHERE sale_date = o_date AND product = o_prod; if SQL%NOTFOUND then ---------- ④ INSERT INTO SALES_PER_DATE VALUES(o_date, o_prod, :NEW.qty, :NEW.amount); end if; END; /

SUMMARY_SALES Trigger의 처리절차를 설명하면 다음과 같다.

① Trigger를 선언한다.CREATE OR REPLACE Trigger SUMMARY_SALES : Trigger 선언문AFTER INSERT : 레코드가 입력이 된 후 Trigger 발생 ON ORDER_LIST : ORDER_LIST 테이블에 Trigger 설정FOR EACH ROW : 각 ROW마다 Trigger 적용 ② o_date(주문일자), o_prod(주문상품) 값을 저장할 변수를 선언하고, 신규로 입력된 데이터를 저장한다. : NEW는 신규로 입력된 레코드의 정보를 가지고 있는 구조체 : OLD는 수정, 삭제되기 전의 레코드를 가지고 있는 구조체 [표 Ⅱ-2-17] 참조 ③ 먼저 입력된 주문 내역의 주문 일자와 주문 상품을 기준으로 SALES_PER_DATE 테이블에 업데이트한다. ④처리 결과가 SQL%NOTFOUND이면 해당 주문 일자의 주문 상품 실적이 존재하지 않으며, SALES_ PER_DATE 테이블에 새로운 집계 데이터를 입력한다.

[표 Ⅱ-2-17] Trigger에서 사용하는 레코드 구조체 비교(1)

구분	:OLD	:NEW
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

[예제] SQL Server CREATE Trigger dbo.SUMMARY_SALES ---------- ① ON ORDER_LIST AFTER INSERT AS DECLARE @o_date DATETIME,@o_prod INT,@qty int, @amount int BEGIN SELECT @o_date=order_date, @o_prod=product, @qty=qty, @amount=amount FROM inserted ---------- ② UPDATE SALES_PER_DATE ---------- ③ SET qty = qty + @qty, amount = amount + @amount WHERE sale_date = @o_date AND product = @o_prod; IF @@ROWCOUNT=0 ---------- ④ INSERT INTO SALES_PER_DATE VALUES(@o_date, @o_prod, @qty, @amount) END

SUMMARY_SALES Trigger의 처리절차를 설명하면 다음과 같다.

① Trigger를 선언한다.CREATE Trigger SUMMARY_SALES : Trigger 선언문ON ORDER_LIST : ORDER_LIST 테이블에 Trigger 설정AFTER INSERT : 레코드가 입력이 된 후 Trigger 발생 ② o_date(주문일자), o_prod(주문상품), qty(수량), amount(금액) 값을 저장할 변수를 선언하고, 신규로 입력된 데이터를 저장한다.inserted는 신규로 입력된 레코드의 정보를 가지고 있는 구조체|deleted는 수정, 삭제되기 전의 레코드를 가지고 있는 구조체. [표 Ⅱ-2-18] 참조 ③ 먼저 입력된 주문 내역의 주문 일자와 주문 상품을 기준으로 SALES_PER_DATE 테이블에 업데이트한다. ④ 처리 결과가 0건이면 해당 주문 일자의 주문 상품 실적이 존재하지 않으며, SALES_PER_DATE 테이블에 새로운 집계 데이터를 입력한다.

[표 Ⅱ-2-18] Trigger에서 사용하는 레코드 구조체 비교(2)

구분	deleted	inserted
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

[예제] ORDER_LIST 테이블에 주문 정보를 입력한다.

[예제] Oracle SQL> SELECT * FROM ORDER_LIST; 선택된 레코드가 없다. SQL> SELECT * FROM SALES_PER_DATE; 선택된 레코드가 없다. SQL> INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000); 1개의 행이 만들어졌다. SQL> COMMIT; 커밋이 완료되었다.

[예제] SQL Server SELECT * FROM ORDER_LIST; 선택된 레코드가 없다. SELECT * FROM SALES_PER_DATE; 선택된 레코드가 없다. INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000); 1개의 행이 만들어졌다.

[예제] 주문 정보와 판매 집?.

[실행 결과] SQL> SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT --------- -------- -------- ------- 20120901 MONOPACK 10 300000 SQL> SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- -------- -------- -------- 20120901 MONOPACK 10 300000

[예제] 다시 한 번 같은 데이터를 입력해보고, 두 테이블의 데이터를 확인한다.

[실행 결과] Oracle SQL> INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000); 1개의 행이 만들어졌다. SQL> COMMIT; 커밋이 완료되었다. SQL> SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT --------- ---------- ------ ------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 SQL> SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- --------- ----- ------- 20120901 MONOPACK 30 900000

[실행 결과] SQL Server INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000); 1개의 행이 만들어졌다. SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT --------- --------- ----- -------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- --------- ---- -------- 20120901 MONOPACK 30 900000

[예제] 이번에는 다른 상품으로 주문 데이터를 입력한 후 두 테이블의 결과를 조회해 보고 트랜잭션을 ROLLBACK 수행한다. 판매 데이터의 입력 취소가 일어나면, 주문 정보 테이블과 판매 집계 테이블에 동시에 입력(수정) 취소가 일어나는지 확인해본다.

[실행 결과] Oracle SQL> INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000); 1개의 행이 만들어졌다. SQL> SELECT * FROM ORDER_LIST; ORDER_DA PRODUCT QTY AMOUNT -------- -------- ------ ------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 20120901 MULTIPACK 10 300000 SQL> SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- -------- ------ ------- 20120901 MONOPACK 30 900000 20120901 MULTIPACK 10 300000 SQL> ROLLBACK; 롤백이 완료되었다. SQL> SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT -------- -------- ------ ------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 SQL> SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- --------- ------ ------- 20120901 MONOPACK 30 900000

[실행 결과] SQL Server BEGIN TRAN INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000); 1개의 행이 만들어졌다. SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT --------- --------- ------ ------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 20120901 MULTIPACK 10 300000 SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- --------- ------ ------- 20120901 MONOPACK 30 900000 20120901 MULTIPACK 10 300000 ROLLBACK; 롤백이 완료되었다. SELECT * FROM ORDER_LIST; ORDER_DATG PRODUCT QTY AMOUNT --------- -------- ------ ------- 20120901 MONOPACK 10 300000 20120901 MONOPACK 20 600000 SELECT * FROM SALES_PER_DATE; SALE_DATG PRODUCT QTY AMOUNT -------- -------- ------ ------- 20120901 MONOPACK 30 900000

ROLLBACK을 하면 하나의 트랜잭션이 취소가 되어 Trigger로 입력된 정보까지 하나의 트랜잭션으로 인식하여 두 테이블 모두 입력 취소가 되는 것을 보여주고 있다. Trigger는 데이터베이스에 의해 자동 호출되지만 결국 INSERT, UPDATE, DELETE 문과 하나의 트랜잭션 안에서 일어나는 일련의 작업들이라 할 수 있다. Trigger는 데이터베이스 보안의 적용, 유효하지 않은 트랜잭션의 예방, 업무 규칙 자동 적용 제공 등에 사용될 수 있다.

7. 프로시저와 트리거의 차이점

프로시저는 BEGIN ~ END 절 내에 COMMIT, ROLLBACK과 같은 트랜잭션 종료 명령어를 사용할 수 있지만, 데이터베이스 트리거는 BEGIN ~ END 절 내에 사용할 수 없다.

[표 Ⅱ-2-19] 프로시저와 트리거의 차이점

프로시저	트리거
CREATE Procedure 문법사용	CREATE Trigger 문법사용
EXECUTE 명령어로 실행	생성 후 자동으로 실행
COMMIT, ROLLBACK 실행 가능	COMMIT, ROLLBACK 실행 안됨