# Oracle Database for DBA

## Section 2 : SQL

## 6. Query

## 1. Subqueries

*Introduction to Oracle 9i SQL : Volume Ⅰ (Chapter 6)*

## 2. Advanced Subqueries

*Introduction to Oracle 9i SQL : Volume Ⅱ (Chapter 18)*

- ◆ Scalar Subquery (Chapter 18: 11, 12)
- ◆ Correlated Subquery (Chapter 18: 14~20)
  Correlated UPDATE (Chapter 18: 21~22)
  Correlated DELETE (Chapter 18: 24, 25)

## 3. Using SET Operators

*Introduction to Oracle 9i SQL : Volume Ⅱ (Chapter 15)*

## 4. Enhancements to the GROUP BY Clause

*Introduction to Oracle 9i SQL : Volume Ⅱ (Chapter 17)*

- ◆ GROUP BY with ROLLUP and CUBE Operators (Chapter 17: 6)
  ROLLUP Operator (Chapter 17: 7~8)
  CUBE Operator (Chapter 17: 9~10)
  GROUPING Function (Chapter 17: 11~12)
  GROUPING SETs (Chapter 17: 13~16)
  Composite Columns (Chapter 17: 17~20)
  Concatenated Groupings (Chapter 17: 21~22)

## 5. Hierarchical Retrieval

*Introduction to Oracle 9i SQL : Volume Ⅱ (Chapter 19)*

- ◆ Hierarchical Queries (Chapter 19)

## 6. Analytic Function

*(출처: ㈜엔코아 http://www.dator.co.kr ) Edited By SaintJoe*

### 1) Analytic Function 의 소개 및 수행원리

- **Analytic Function 의 소개**

초기 대부분의 RDBMS 는 집합적인 개념에 충실하여 만들어 졌기 때문에 집합적인 개념에 위배되는 처리는 표준 SQL 로 처리가 불가능하였다. 그래서 이러한 작업은 프로그램 로직으로 처리 하거나 데이터의 복제 등 다양한 응용 고난도 SQL(대용량 데이터베이스 2 권 참조)을 활용하여 처리 해야만 했다.

그러나, 프로그램에서 로직으로 처리할 경우 집합 개념의 RDB 시스템에서 심각한 성능 저하를 유발할 수 있다는 점을 "고성능 DB 구축을 위한 핵심요소의 이해"를 읽으신 분들은 이해하실 수 있을 것이다. 즉 고가의 고성능 RDB 를 구입해 놓고 DBMS 를 단지 Data 저장소로만 사용하는 방법이다. 프로그램에서 IF..then..else 를 이용하여 필요하면 그때 그때 Data 저장소(RDB)에서 Data 를 읽어와 상황에 따라 처리 및 판단을 한 후 조건에 맞으면 저장해 놓고 마지막에 모든 값을 조합하여 최종 결과를 Return 하는 형식이 될 것이다.

특히 비즈니스 분석작업에서 위와 같은 방법으로 처리할 경우 어떤 결과를 초래 하겠는가? 분석작업에 필요한 Data 라면 범위가 일부분에만 국한되지는 않을 것이다. 때에 따라서 범위를 정하여 분석하는 경우도 있고, 전체를 대상으로 분석하는 경우도 발생하겠지만 거의 대부분은 넓은 범위의 Data 가 대상이 된다는 점이다. 이런 넓은 범위의 다량의 Data 를 반복적으로 읽어와서 프로그램 로직으로 처리할 경우 이미 성능을 논할 단계는 넘어간 것이다.

가령 100 만 건의 고객 Data 를 직업별로 분류하고 연봉이 높은 순으로 몇 개의 Level 을 정하여 각 직업별 Level 별 평균 연봉 및 고객 수 및 각 직업별 평균연봉 및 고객 수를 보고자 한다고 100 만건을 건 별로 읽어서 직업이 무엇인지 구분하고 연봉에 해당 고객의 연봉을 저장하고 어느 Level 에 속하는지 구분한 후, 다시 한 건의 고객을 읽은 후 분류 및 기존에 이미 읽은 Data 에 합계를 수행하고, 건수를 기록하고 다시 또 한 건의 고객을 읽는 작업을 100 만번 수행해야 한다. 그 다음에 각 직업별로 연봉을 비교하여 순위를 정한 후 이 순위에 따라 차례대로 나열하는 작업을 수행해야 한다. 만약에 연봉이 아니라 한 고객 당 평균 약 10 회의 매출을 발생시킨 상황에서 매출 금액이 높은 직업별로 보고자 한다면 또 어떻게 할 것인가. 위와 같은 처리가 다시 고객 건 별로 약 10 회씩 반복 처리를 하게 될 것이다. 즉 답을 구하는 데는 문제가 없지만 효율성 측면에서는 재고(再考)해야만 하는 상황이 되는 것이다.

그러면 성능 개선을 위해 RDB 의 집합개념을 이용하여 고성능 SQL 로 해결하면 어떻게 되는가? 해당하는 전체 집합을 정한 후, 이 전체 집합을 보고자 하는 Level 로 Group By 를 하여 나누어 주고 Decode 를 이용하여 분리한 통계를 만들고 다시 직업별 통계를 구하기 위해 원 집합을 복제하여 직업별로 Grouping 작업을 수행한 후 두 집합을 결합하는 하나의 SQL 을 만들어 DBMS 에게 수행하라고 명령(Query)하는 것이다. 이 경우 DBMS 가 해당하는 대상이 되는 Data 를 읽어서 Group By 를 수행하고 조합한 후 다시 순위를 매겨 최종 결과만을 프로그램에 Return 하게 된다. – 여기선 비교적 간단한 분석/통계 작업의 예를 들었지만 위의 예에다가 Group(직업)별 순위를 매기고 순위에 따라 고객 신용등급을 다르게 적용하는 등 몇 가지 응용이 추가된다면 Group Serial 등 다양한 기법의 고난도 SQL 을 적용하게 될 것이다. -

실제 하는 일을 DBMS 가 했는가, DBMS 에서 읽어와서 프로그램의 로직에서 처리했는가

만의 차이가 있을 뿐 하고자 한 행위는 동일하다. 그러나 첫 번째 방법은 System 의 성능에 지대한 영향을 미칠 것이 자명하고, 두 번째 방법은 SQL 을 구사하는데 고난도의 기술이 필요하며 유지보수를 하는데 있어서도 개발자가 적용한 SQL 의 정확한 개념을 이해해야만 원활하게 작업할 수 있을 것이다.

그러면 이 두 가지 문제를 동시에 해결할 수는 없을까? 그룹 내 순위를 정하기 위해 복잡하게 SQL 을 구사한 부분을 하나의 명령으로 DBMS 가 처리해 줄 수는 없을까? 집합개념에서는 처리가 되지 않는 각 Row 간의 값을 비교할 수는 없을까? 즉 사용자의 비즈니스 요구사항이 더욱 더 복잡해 지면서 구현이 용이하고 성능도 향상시킬 수 있는 강력한 SQL 의 필요성이 절실하게 요구되었고 이러한 필요에 따라 Red Brick 은 DATA ANALYSIS 나 DSS(DECISION-SUPPORT SYSTEM)에 적합한 다양하고 강력한 기능을 가진 SQL 을 제안하였는데, 새로운 제안에는 집합적 개념인 표준 SQL 에서 처리가 어려워 절차적으로 처리할 수 밖에 없었던 비즈니스 분석 요구를 수용하기 위해 cume, MovingAvg(n), MovingSum(s), Rank… When, Ratio To Report, Tertile, Create Macro 와 같은 많은 functions 들의 지원을 포함하고 있으며 이는 집합개념에서 수용하지 못하였던 포인터(Pointer)와 오프셋(Offset)의 개념을 추가 시킨 것으로 이 SQL 을 RISQL(Red Brick Intelligent SQL)이라 한다. 여기서 추가된 개념들이 바로 Analytic Function 이다.

Analytic Function 을 지원하는 RDBMS 를 사용하는 경우 Self-Join 또는 클라이언트 프로그램의 절차적 로직으로 표현한 것 또는 SQL 로 표현하기 위해 고난도의 여러 기법을 적용하였던 것을 Native SQL 내에서 하나의 명령어로 바로 적용할 수 있으므로 조인이나 클라이언트 프로그램의 Overhead 를 줄임으로써 Query 속도를 향상시킬 수 있고, 개발자가 명백하고 간결한 SQL 로 복잡한 분석작업을 수행할 수 있으며, 개발 및 유지보수가 편하기 때문에 생산력을 향상시킬 수 있다.

또한, 기존 SQL Syntax 를 그대로 따르기 때문에 기존 Standard SQL 을 사용하던 개발자/운영자의 이해가 빠르며 적용하기 쉽고 ANSI SQL 채택으로 향후 다양한 소프트웨어에 적용이 될 것이므로 때문에 표준화에도 유리한 장점이 있으므로 그 활용 정도가 점차 확대될 것이다.

현재 모든 상용 DBMS 가 Analytic Function 을 제공하는 것이 아니다. 본 글은 Oracle 사의 DBMS 가 Support 하는 Analytic Function 을 기준으로 간단한 수행원리 및 각 기능별 활용사례를 소개하고자 한다.

- **Analytic Function 의 수행원리**

**Analytic Function 의 수행 절차 (Processing Order)**

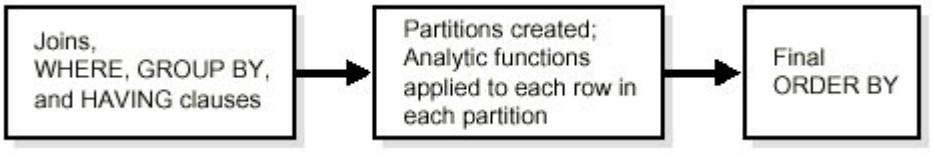
**- 1 단계 (General Processing)**
Join, Where, Group By and Having 등의 기존 Query Processing 수행단계로서 기존 Standard SQL 이 수행되는 동일한 원리에 의해서 대상 집합을 추출하는 단계이다.

**- 2 단계 (Analytic Function Applying)**
1 단계 결과를 가지고 실제 Analytic Function 이 적용되어 필요한 계산을 수행하는 단계, 즉 대상집합을 필요한 몇 개의 Group 으로 분리하고 순위를 결정하며, 그룹 순위를 기준으로 명령된 계산을 수행하는 단계이다.

이 단계에서 내부적으로 적용되는 세부 메커니즘은 다음과 같이 수행된다.

① 대상집합을 Analytic Function 이 적용되어야 할 각 Group 으로 나눈다. (Partitioning)
② 각 Partition 내의 집합에 속한 개체에 조건에 따른 순위를 결정한다.
③ Pointer 와 Off-Set 개념을 적용하여 각 Row 간에 필요한 계산을 수행한다.

**- 3 단계 (Order By Processing – Optional)**
Query 에 Order By 절이 있다면 최종결과에 대한 Ordering 을 행하는 단계




**Analytic Function 를 사용했을 때 SELECT-LIST 의 출력순서**

partition by column 을 기준으로 1 차 정렬이 되고 order by 절이 있다면, 해당 partition 내에서 order by column 으로 다시 정렬이 되어 출력된다.

한 SELECT-LIST 에 partition by column 이나 order by column 이 혼재 되어 있을 경우, SELECT-LIST 의 제일 처음으로 기술된 analytic function 의 정렬순서를 따르게 된다.

```
select deptno, hiredate, sal,
       sum(sal) over( partition by deptno order by hiredate) cavg1,
       sum(sal) over( partition by deptno order by hiredate desc) cavg2
  from scott.emp      → (A)
```

vs.

```
select deptno,hiredate,sal,
       sum(sal) over( partition by deptno order by hiredate desc) cavg2,
       sum(sal) over( partition by deptno order by hiredate) cavg1
  from scott.emp    → (B)
```

(A)와 (B)는 집합적으로는 같은 결과를 내나, 출력순서는 서로 다르다.
(A)는 deptno 별 hiredate 순으로 출력되나 (B)는 deptno 별 hiredate 의 역순으로 출력된다.

**Analytic Function 의 3 요소**

① Result Set Partitions

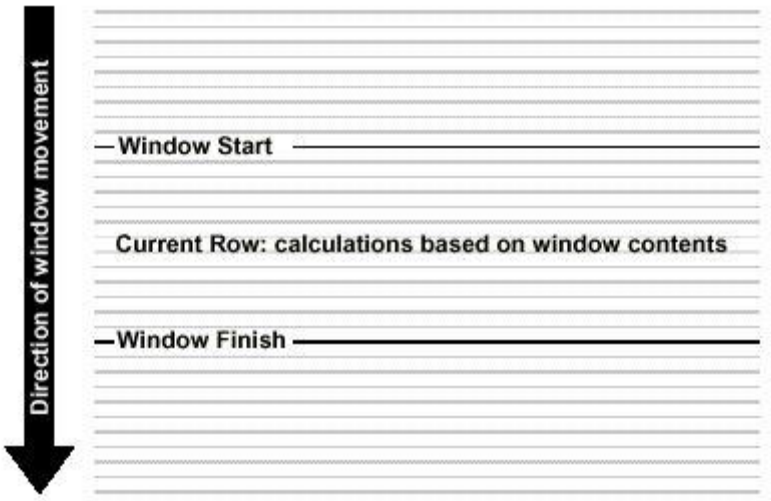
Query Processing with Analytic Function 의 1 단계 수행결과를 Column 이나 Expression 을 기준으로 Grouping 한 것. 1 단계 수행결과 전체가 하나의 Partition 에 속할 수도 있고, 적은 Rows 를 가진 여러 개의 작은 Partition 으로 쪼개질 수도 있다. 그러나, 한 Row 는 반드시 하나의 Partition 에 속한다.

② (Sliding) Window

Current Row 에 대한 Analytic Calculation 수행의 대상이 되는 Row 의 범위(Range), Window 는 Current Row 를 기준으로 하나의 Partition 내에서 Sliding 하면, 반드시 Starting Row 와 Ending Row 를 가진다. Window Size 는 Partition 전체가 될 수도 있고 Partition 의 부분범위가 될 수도 있으나 하나의 Partition 을 넘을 수는 없다. Partition 의 부분범위로서 Window Size 를 정할 때는 Physical Number of Rows 로 정할 수도 있고 Logical Interval 로 정할 수도 있다.

③ Current Row

모든 Analytic Function 의 적용은 항상 Partition 내의 Current Row 를 기준으로 수행된다. Current Row 는 항상 Window 의 Start 와 End 를 결정하는 기준(Reference Point)으로서 역할을 하므로 Current Row 가 없는 Window 는 존재하지 않는다.



**Analytic Function 의 종류**
Analytic Function 은 그 성격에 따라 다음과 같이 4 개의 Family Group 으로 구분된다.

① **Ranking Family**
대상 집합에 대하여 특정 Column(s) 기준으로 순위나 등급을 매기는 Analytic Function 류로서 다음과 같은 종류가 있다.

**RANK(), DENSE_RANK(), CUME_DIST(), PERCENT_RANK(), NTILE(), ROW_NUMBER()**

**<Syntax>**
**{*Ranking Families*} OVER ( [PARTITION BY <value expression1> [, ...]]**
**ORDER BY <value expression2> [collate clause]**
**[ASC|DESC] [NULLS FIRST|NULLS LAST] [, ...] )**

② **Window Aggregate Family**
Current Row 를 기준으로 지정된 Window 내의 Rows 를 대상으로 집단화(Aggregation)를 수행하여 여러 가지 유용한 집계정보(Running Summary, Moving Average)를 구하는 Analytic Function 류이며 다음과 같은 종류가 있다.

**SUM, AVG, MIN, MAX, STDDEV, VARIANCE, COUNT, FIRST_VALUE, LAST_VALUE**

**<Syntax>**
**{*Window Aggregate Families*} ({<value expression1>|*})**
**OVER ( [PARTITION BY <value expression2>[,...]]**
**ORDER BY <value expression3> [collate clause>]**
**[ASC|DESC] [NULLS FIRST|NULLS LAST] [,...]**
**ROWS|RANGE {{UNBOUNDED PRECEDING|<value expression4> PRECEDING}**
**|BETWEEN UNBOUNDED PRECEDING|<value expression4> PRECEDING}**
**AND{CURRENT ROW | <value expression4> FOLLOWING}} )**

③ **Reporting Aggregate Family**
서로 다른 두 가지의 Aggregation Level 을 비교하고 하는 목적으로 사용하는 Analytic Function 으로서 다음과 같은 종류가 있다.

**SUM, AVG, MAX, MIN, COUNT, STDDEV, VARIANCE**

**<Syntax>**
**{*Reporting Aggregate Families*} ([ALL|DISTINCT] {<value expression1>|*})**
**OVER ([PARTITION BY <value expression2>[,...]])**

④ LEAD/LAG Family
서로 다른 두 Row 값을 비교하기 위한 Analytic function.

**LEAD( ), LAG( )**

**<Syntax>**
**{LAG|LEAD} (<value expression1>, [<offset> [, <default>]])**
**OVER ([PARTITION BY <value expression2>[,...]]**
**ORDER BY <value expression3> [collate clause>]**
**[ASC|DESC] [NULLS FIRST| NULLS LAST] [,...])**

### 2) Ranking Family 의 소개 및 활용 사례

- **Ranking Family 의 특징**

  Ranking Family 에 속하는 Analytic Function 은 대상 집합에 대하여 특정 Column(s)을 기준으로 순위나 등급을 부여하는 것으로 다음과 같은 특징을 지니고 있다.

  ① 오름차순 또는 내림차순으로 순위나 등급을 부여할 수 있다

  ② 오름차순, 내림차순과 관계없이 NULL 은 순위의 가장 처음 또는 마지막으로 강제 처리 가능하다.

  ③ Rank functions 은 각 파티션마다 초기화된다.

  ④ 순위 또는 등급은 GROUP BY CUBE 와 ROLLUP 절마다 초기화된다.

  **<Functions>**
  **RANK(), DENSE_RANK(), CUME_DIST(), PERCENT_RANK(), NTILE(), ROW_NUMBER()**

  **<Syntax>**
  **{*Ranking Families*} OVER ( [PARTITION BY <value expression1> [, ...]]**
  **ORDER BY <value expression2> [collate clause]**
  **[ASC|DESC] [NULLS FIRST|NULLS LAST] [, ...] )**

  Windowing 절을 사용하려 할 때는 반드시 order by 절을 사용해야 한다.

- **Ranking Family 의 종류**

**① RANK( )**

RANK 함수는 각 Row 마다 순위를 매겨주는 함수로 각 PARTITION 내에서 ORDER BY 절에 명시된 대로 정렬한 후의 순위를 의미하고 1 부터 시작하여 동일한 값은 동일한 순위를 가지며, 동일한 순위의 수만큼 다음 순위는 건너뛴다.


[그림 1]은 "판매실적에 의한 제품의 순위를 각 지역별과 전체 지역에서 각각 부여하라"는 RANK( )함수의 활용한 예로 GROUP BY 절과 RANK( )함수가 같이 사용되었다

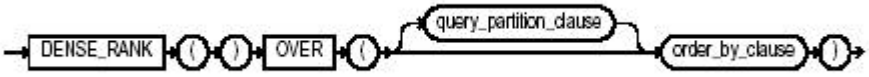
```
SELECT   r_regionkey, p_productkey, sum(s_amount),
         RANK() OVER (PARTITION BY r_regionkey ORDER BY sum(s_amount) DESC)
                                                    AS rank_of_product_per_region,
         RANK() OVER (ORDER BY sum(s_amount) DESC)
                                                    AS rank_of_product_total
  FROM product, region, sales
 WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
 GROUP BY r_regionkey, p_productkey;
```

| r_region key | p_product key | s_amount | rank_of_product _per_region | rank_of_ product_total |
|---|---|---|---|---|
| east | shoes | 130 | 1 | 1 |
| east | jackets | 95 | 2 | 4 |
| east | shirts | 80 | 3 | 6 |
| east | sweaters | 75 | 4 | 7 |
| east | t-shirts | 60 | 5 | 11 |
| east | ties | 50 | 6 | 12 |
| east | pants | 20 | 7 | 14 |
| west | shoes | 100 | 1 | 2 |
| west | jackets | 99 | 2 | 3 |
| west | t-shirts | 89 | 3 | 5 |
| west | sweaters | 75 | 4 | 7 |
| west | shirts | 75 | 4 | 7 |
| west | ties | 66 | 6 | 10 |
| west | pants | 45 | 7 | 13 |

[그림 1] 동일 영업실적에 대해서는 동일한 순위를 부여하고 있고 다음 영업실적은 동일 순위의 수만큼을 건너 뛴 순위가 부여되었음을 확인할 수 있다

② **DENSE_RANK( )**

DENSE_RANK( )는 Rank( )와 유사한 함수로 ORDER BY 절에 사용된 Column 이나 표현식에 대하여 순위를 부여하는데 RANK( )와 달리 동일 순위 다음의 순위는 동일 순위의 수와 상관없이 1 증가된 값을 돌려준다.



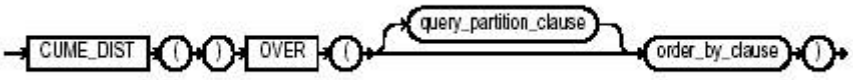

[그림 2]는 RANK( )와 DENSE_RANK( )가 어떻게 다른 것인지를 보여주고 있다.

| person | amount | RANK | DENSE_RANK |
|---|---|---|---|
| Adams | 100 | 1 | 1 |
| Baker | 100 | 1 | 1 |
| Connors | 89 | 3 | 2 |
| Davis | 75 | 4 | 3 |
| Edwards | 75 | 4 | 3 |
| Fitzhugh | 66 | 6 | 4 |
| Garibaldi | 45 | 7 | 5 |

[그림 2] RANK()와 DENSE_RANK()의 비교

③ **CUME_DIST( )** : Cumulative Distribution Function

PARTITION BY 에 의해 나누어진 그룹별로 각 row 를 ORDER BY 절에 명시된 순서로 정렬한 후 그룹별 상대적인 위치(누적된 분산정도)를 구한다. 상대적인 위치는 구하고자 하는 값보다 작거나 같은 값을 가진 ROW 수를 그룹 내 총 ROW 수로 나눈 것을 의미하며 결과 값의 범위는 0 보다 크고 1 보다 작거나 같다.



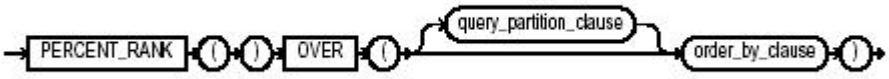

```
SELECT   r_regionkey, p_productkey, SUM(s_amount) as s_amount,
         CUME_DIST() OVER (PARTITION BY r_regionkey ORDER BY SUM(s_amount))
                                                    AS cume_dist_per_region
  FROM region, product, sales
 WHERE r_regionkey = s_regionkey AND p_productkey = s_productkey
 GROUP BY r_regionkey, p_productkey
 ORDER BY r_regionkey, s_amount DESC;
```

| r_regionkey | p_productkey | s_amount | cume_dist_per_region | |
|---|---|---|---|---|
| east | shoes | 130 | 1.00 | → East Group의 7 Rows 중 7/7 |
| east | jackets | 95 | 0.86 | |
| east | shirts | 80 | 0.71 | |
| east | sweaters | 75 | 0.57 | |
| east | t-shirts | 60 | 0.43 | |
| east | ties | 50 | 0.29 | → East Group의 7 Rows 중 2/7 |
| east | pants | 20 | 0.14 | → East Group의 7 Rows 중 1/7 |
| west | shoes | 100 | 1.00 | |
| west | jackets | 99 | 0.86 | |
| west | t-shirts | 89 | 0.71 | |
| west | sweaters | 75 | 0.43 | |
| west | shirts | 75 | 0.43 | |
| west | ties | 66 | 0.29 | |
| west | pants | 45 | 0.14 | |

[그림 3] CUME_DIST( )의 활용

**④ PERCENT_RANK( )**

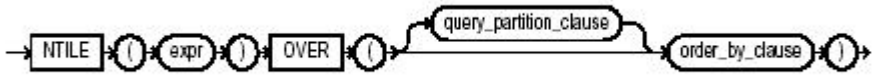
CUME_DIST 와 유사한 함수이나 PARTITION 별 각 row 의 순위 –1 / PARTITION 내의 ROW 의 수를 결과값으로 하며, 결과값 범위는 0 <= 결과값 <= 1 이고 집합의 첫 번째 row 의 PERCENT_RANK 는 항상 0 이 된다.




| Value | RANK( ) | DENSE_RANK( ) | CUME_DIST( ) | PERCENT_RANK( ) |
|---|---|---|---|---|
| 10 | 1 | 1 | 0.25 | 0.00 → (1-1)/(4-1) |
| 20 | 2 | 2 | 0.75 | 0.33 → (2-1)/(4-1) |
| 20 | 2 | 2 | 0.75 | 0.33 → (2-1)/(4-1) |
| 30 | 4 | 3 | 1.00 | 1.00 → (4-1)/(4-1) |

**⑤ NTILE( )**

PARTITION 을 BUCKET 이라 불리는 그룹별로 나누고 PARTITION 내의 각 ROW 등을 BUCKET 에 배치하는 함수로 각 BUCKET 에는 동일한 수의 ROW 가 배치된다. 예를 들어 PARTITION 내에 100 개의 ROW 를 가지고 있고 4 개의 BUCKET 으로 나누는 NTILE(4)를 사용하면 1 개의 BUCKET 당 25 개의 ROW 가 배정된다. 만일 각 PARTION 의 수가 정확하게 분배되지 않을 경우 근사치로 배분한 후 남는 값에 대하여 최초 PARTITION 부터 한 개씩 배분한다. 즉, 만일 103 개의 ROW 에 대하여 NTILE(5)를 적용하면 첫 번째 BUCKET 부터 세 번째까지는 21 개의 ROW 가, 나머지는 20 개의 ROW 가 배치된다.




[그림 5]는 "제품별 판매량을 구하고, 이를 4 등급으로 나누어, 다시 등급 내에서 순위를 매겨라"는 질의를 NTILE()함수를 이용하여 구현한 것이다.

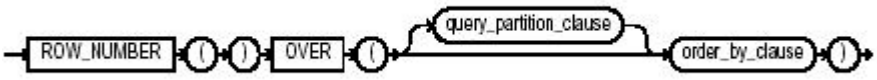
```
SELECT   p_productkey, sum(s_amount) AS sum_s_amount,
           NTILE(4) over (ORDER BY sum(s_amount) DESC) as 4_tile,
           RANK() OVER (PARTITION BY NTILE(4) over (ORDER BY sum(s_amount) DESC)
                                          ORDER BY sum_s_amount DESC) AS rank_in_quartile
  FROM product, sales
 WHERE p_productkey = s_productkey
 GROUP BY p_productkey ;
```

| p_productkey | sum_s_amount | 4_tile | rank_in_quartile |
|---|---|---|---|
| shoes | 100 | 1 | 1 |
| jackets | 90 | 1 | 2 |
| shirts | 89 | 2 | 1 |
| sweaters | 75 | 2 | 2 |
| shirts | 75 | 3 | 1 |
| ties | 66 | 3 | 2 |
| pants | 45 | 4 | 1 |
| socks | 45 | 4 | 1 |

[그림 5] NTILE( ) 활용

**⑥ ROW_NUMBER( )**

ROW_NUMBER()는 각 PARTITION 내에서 ORDER BY 절에 의해 정렬된 순서로 유일한 값을 돌려주는 함수로 ROWNUM 과는 관계가 없다.




[그림 6]은 "제품별 판매량의 내림차순으로 unique 한 일련번호를 부여하라"는 질의에 ROW_NUMBER()를 활용한 예이다.

```
SELECT   p_productkey, sum(s_amount),
           ROW_NUMBER() (ORDER BY sum(s_amount) DESC NULLS LAST) AS srnum
  FROM product, sales
 WHERE p_productkey = s_productkeyGROUP BY p_productkey;
```

| p_productkey | s_amount | srnum |
|---|---|---|
| shoes | 100 | 1 |
| jackets | 90 | 2 |
| shirts | 89 | 3 |
| t-shirts | 84 | 4 |
| sweaters | 75 | 5 |
| jeans | 75 | 6 |
| ties | 75 | 7 |
| pants | 69 | 8 |
| belts | 56 | 9 |
| socks | 45 | 10 |
| suits | NULL | 11 |

[그림 6] ROW_NUMBER( ) 활용

- **Ranking Family 의 활용 사례**

이 절에서는 위에서 소개된 Ranking Family Analytic Function 의 실제업무에 이용 가능한 활용사례에 대해 살펴보고자 한다. 아래에 소개된 예들은 ORACLE 8i Version 을 중심으로 작성되었으나 UNIX Version 의 IBM UDB 7 에서도 OLAP Function 이라하여 동일한 문법의 함수를 지원하므로 사용 가능할 것으로 보여진다. 다만 DBMS 별로 지원하는 함수의 종류에는 차이가 있으면 RANK 계열 함수의 경우 UDB 7 의 경우 RANK, DENSE_RANK, ROW_NUMBER 가 지원됨을 확인하였다. 다른 DBMS 에서도 Analytic Function 이 SQL 1999 의 표준으로 채택된 만큼 이미 사용 가능하거나 아닐지라도 가까운 시일 내에 지원되리라고 보여진다.

**① 달력상의 주 별로 순번을 부여하는 SQL 을 작성하시오. (ROW_NUMBER 활용사례 1)**

아래 그림과 같은 결과가 나오도록 SQL 을 작성하시오. 여기에서 주의할 점은 달(월)이 바뀐 경우이다. 즉 달력상의 1 주(일 ~ 월)를 하나의 Grouping 으로 해야 한다. 또한 달이 바뀔 경우에는 이를 무시하고 다시 순번을 시작해야 한다.

| 일자 | 순번 | 일자 | 순번 |
|---|---|---|---|
| 20020101 | 1 | **20020201** | 1 |
| 20020104 | 2 | **20020203** | 1 |
| 20020105 | 3 | 20020207 | 2 |
| 20020106 | 1 | 20020209 | 3 |
| 20020107 | 2 | 20020210 | 1 |
| 20020111 | 3 | 20020211 | 2 |
| 20020112 | 4 | 20020221 | 1 |
| 20020113 | 1 | 20020223 | 2 |
| 20020114 | 2 | 20020224 | 1 |
| 20020117 | 3 | 20020225 | 2 |
| ..... | ..... | ..... | ..... |

```
SQL> SELECT 일자,
            ROW_NUMBER() OVER (PARTITION BY substr(일자, 1, 6) ||
                                            to_char(to_date(일자, 'yyyymmdd') -
                                            to_char(to_date(일자, 'yyyymmdd'),'d')+1, 'yyyymmdd')
                                           ORDER BY 일자) 순번
       FROM SAMPLE1 ;
```

**② 아래 그림의 월별 급여 집계테이블에서 2001 년 1 월의 각 부서별 급여액 순으로 상위 2 위에 해당 하는 사원 1 명의 이름, 급여액을 구하는 SQL 을 작성하시오.**
**(ROW_NUMBER 활용사례 2)**

단, 급여액이 동일할 경우 성명 순으로 한다.
(즉, 동률의 급여금액이 존재하더라도 부서별 급여금액 역순, 이름순으로 정렬하여 그 순서에 따른 2 위(두 번째) 를 의미)

[월별 급여집계 테이블]

| 부서 | 성명 | 년월 | 급여액 |
|---|---|---|---|
| 총무부 | 홍길동 | 200101 | 200 |
| 총무부 | 임꺽정 | 200101 | 250 |
| 총무부 | 장길산 | 200101 | 100 |
| 총무부 | 최고봉 | 200101 | 200 |
| 총무부 | 황비홍 | 200101 | 120 |
| 영업부 | 궁해 | 200101 | 300 |
| 영업부 | 왕곤 | 200101 | 310 |
| 영업부 | 견온 | 200101 | 220 |
| 생산부 | 박임자 | 200101 | 350 |
| 생산부 | 전본인 | 200101 | 300 |
| 생산부 | 노친구 | 200101 | 300 |
| 생산부 | 김갱제 | 200101 | 230 |

[결과 집합]

| 부서 | 성명 | 급여액 |
|---|---|---|
| 총무부 | 최고봉 | 200 |
| 영업부 | 궁해 | 300 |
| 생산부 | 노친구 | 300 |

```
SQL> SELECT 부서, 성명, 급여액
       FROM
            (SELECT 부서, 성명, 급여액,
                    ROW_NUMBER() OVER (PARTITION BY 부서
                                       ORDER BY 급여액 DESC, 성명) SEQ
             FROM 월별급여테이블
            WHERE 년월 = '200101' )
     WHERE SEQ = 2 ;
```

※ 이 경우 Analytic Function 중 RANK 또는 DENSE_RANK 를 사용하면 문제에서 원하는 (금액역순+성명)의 순서 중 2 번째를 구하는데 1 위에 해당하는 동률이 있을 경우 답이 틀리게 됩니다

**③ 아래 테이블의 데이터를 참조하여 결과집합을 구하는 SQL 을 작성하시오.**
**(RANK 활용사례 1)**

CO001T 는 코드 테이블로 하나의 MAJOR 코드에 대하여 최대 3 개의 MINOR 코드를 가질 수 있다.
우리가 보고자 하는 것은 각 MAJOR 코드에 대하여 MINOR 코드를 가로로 정렬하여 보고자 한다. 이때 MINOR 코드가 3 개가 아닌 경우에 나머지는 공백으로 둔다.

[코드 테이블인 CO001T 의 데이터 구조]

| Major | Minor |
|---|---|
| 영업 | 가마니 |
| 영업 | 나오미 |
| 영업 | 마동탁 |
| 물류 | 사시미 |
| 물류 | 아사달 |
| 회계 | 자몽돌 |
| 회계 | 차이나 |
| 회계 | 카이로 |
| 전산 | 파김치 |

[결과 집합]

| MAJOR | MINOR1 | MINOR2 | MINOR3 |
|---|---|---|---|
| 물류 | 사시미 | 아사달 | |
| 영업 | 가마니 | 나오미 | 마동탁 |
| 전산 | 파김치 | | |
| 회계 | 자몽돌 | 차이나 | 카이로 |

```
SQL> SELECT t1.major,
            max(decode(no, 1, minor)) minor1,
            max(decode(no, 2, minor)) minor2,
            max(decode(no, 3, minor)) minor3
     FROM ( SELECT major, minor,
                   RANK() OVER (PARTITION BY major ORDER BY minor ) as no
              FROM co001t
            GROUP BY major, minor ) t1
     GROUP BY t1.major
```

**④ 두 개의 상품이력 테이블을 선분 BETWEEN 조인과 데이터의 복제 방법을 활용하여 아래의 결과 집합을 구하는 SQL 을 작성하시오. (ROW_NUMBER 활용사례 3)**

(주의사항: 결과집합에는 가입계약 별로 할인전화번호가 반드시 3 개가 존재해야만 한다. 따라서 상품이력에는 존재하고 상품 별 할인전화번호가 없는 경우는 계약 별 상품이력에 있는 전화번호로 3 개의 전화번호를 생성 해야만 한다.)

[계약번호 별 상품이력]

| 계약번호 | 상품코드 | 전화번호 | 사용시작일 | 사용종료일 |
|---|---|---|---|---|
| 200100001 | FAMILY01 | 01201111212 | 20010107 | 20010502 |
| 200100002 | FAMILY02 | 01201112020 | 20010301 | 20010707 |
| 200100003 | COUPLE01 | 01208711212 | 20010601 | 99991231 |
| 200100002 | FAMILY01 | 01208721212 | 20010707 | 99991231 |
| 200100001 | COUPLE02 | 01208731212 | 20010502 | 99991231 |

[상품별 할인 전화번호 이력]

| 계약번호 | 등록전화번호 | 사용시작일 | 사용종료일 |
|---|---|---|---|
| 200100001 | 00022341212 | 20010107 | 99991231 |
| 200100001 | 01190661868 | 20010107 | 99991231 |
| 200100001 | 01608711313 | 20010107 | 20010502 |
| 200100001 | 00024527979 | 20010502 | 99991231 |
| 200100002 | 00025441868 | 20010301 | 99991231 |
| 200100002 | 01720661868 | 20010301 | 99991231 |
| 200100002 | 01608713313 | 20010301 | 99991231 |
| 200100003 | 00322340101 | 20010601 | 99991231 |
| 200100003 | 01198705868 | 20010601 | 99991231 |

[결과 집합]

| 계약번호 | 상품명 | 순번 | 등록전화번호 | 사용시작일 | 사용종료일 |
|---|---|---|---|---|---|
| 200100001 | FAMILY01 | 1 | 00022341212 | 20010107 | 20010502 |
| 200100001 | FAMILY01 | 2 | 01190661868 | 20010107 | 20010502 |
| 200100001 | FAMILY01 | 3 | 01608711313 | 20010107 | 20010502 |
| 200100001 | COUPLE02 | 1 | 00022341212 | 20010502 | 99991231 |
| 200100001 | COUPLE02 | 2 | 01190661868 | 20010502 | 99991231 |
| 200100001 | COUPLE02 | 3 | 00024527979 | 20010502 | 99991231 |
| 200100002 | FAMILY02 | 1 | 00025441868 | 20010301 | 20010707 |
| 200100002 | FAMILY02 | 2 | 01720661868 | 20010301 | 20010707 |
| 200100002 | FAMILY02 | 3 | 01608713313 | 20010301 | 20010707 |
| 200100002 | FAMILY01 | 1 | 00025441868 | 20010107 | 99991231 |
| 200100002 | FAMILY01 | 2 | 01720661868 | 20010107 | 99991231 |
| 200100002 | FAMILY01 | 3 | 01608713313 | 20010107 | 99991231 |
| 200100003 | COUPLE01 | 1 | 00322340101 | 20010601 | 99991231 |
| 200100003 | COUPLE01 | 2 | 01198705868 | 20010601 | 99991231 |
| 200100003 | COUPLE01 | 3 | 01208711212 | 20010601 | 99991231 |

```
SQL) SELECT 계약번호, 상품번호,
            순번+NO-1 순번,
            decode(sign(1-no), -1, 전화번호, 등록전화번호) 등록전화번호,
            GREATEST(X_SDATE, Y_SDATE) 사용시작일,
            LEAST(X_EDATE, Y_EDATE) 사용종료일
       FROM (SELECT x,계약번호, x.상품번호, x.전화번호,
             row_number() over (partition by x.계약번호, x.상품명 order by x.등록전화번호)
                    As 순번,
             count(*) over (partition by x.계약번호, x.상품명 order by x.등록전화번호)
                    As c_no,
                    nvl(y.등록전화번호, x.전화번호) 등록전화번호,
                    x.사용시작일 X_SDATE, x.사용종료일 X_EDATE,
                    nvl(y.사용시작일, '10000101') Y_SDATE,
                    nvl(y.사용종료일, '99991231') Y_EDATE
             FROM 계약별상품이력 x, 상품별할인전화번호이력 y
            WHERE y.계약번호(+) = x.계약번호
              AND y.시작일(+)    < x.종료일   --- 선분 Between 조인
              AND y.종료일(+)    > x.시작일   --- 선분이력이 양편 넣기일 때 ) a,
            copy_t b   --- 복제용 Dummy Table
      WHERE no <= decode(c_no||순번, '11', 3, '22', 2, '21', 1, '31', 1', '32', 1, '33', 1, 0)
```

### 3) Aggregate Family (Reporting)의 소개 및 활용 사례

- **Window Aggregate Family (Reporting) 개념**

  **① 윈도우 집계 유형 (Window Aggregate Family)**

  윈도우 집계 함수는 윈도우를 근간으로 하여 정렬된 로우들의 집합과 그 각각의 로우들에 대한 집계 값을 반환한다. 이 함수들은 집계 함수(SUM, COUNT, MAX, MIN 등등)의 파티션에 속하는 로우들의 Sliding Window 에 대한 계산을 수행하기 위해 윈도우 문법(Window Syntax)을 추가하여 확장한 형태이다.

  **② 보고형 집계 유형 (Reporting Aggregate Family)**

  한 집합 레벨에 대한 집계 값과 다른 집합 레벨에 대한 집계 값의 비교를 통해 분석작업을 하고자 하는 경우가 많다. 예를 들면, 한 사원의 급여와 해당 부서의 평균 급여를 비교하고자 하는 경우나, 그 사원의 급여를 제외한 해당 부서의 평균 급여를 알고자 할 때, 보고용 집계 유형은 셀프 조인을 할 필요 없이 다른 집합 레벨에 대한 집계 값을 계산하여 반환한다. 한 그룹에 대해 하나의 집계 값을 반환하는 집계 함수와 다르게 보고용 집계 함수 (Reporting Aggregate Function)는 윈도우 레벨에서 작업한다. 이 함수는 윈도우 안의 모든 로우에 대해 같은 집계 값을 반환한다. 보고용 집계 함수는 전체 윈도우에 대한 집계 값을 반환하거나 해당 로우를 제외한 전체 윈도우의 집계 값을 계산하여 반환한다. 이 함수들의 대부분은 윈도우 집계 함수(Window Aggregate Function)와 유사하고, 비슷한 기능을 수행한다.

- **Window(Reporting) Aggregate Function 의 Syntax 설명**

  여기서는 Window Aggregate Function 의 기본적인 문법에 대한 설명을 한다. 먼저 Analytic Function 의 집계함수 종류를 살펴보면 기존에 사용하였던 모든 집계 함수(SUM, COUNT, AVG, MIN, MAX, STDDEV, VARIANCE) 들을 윈도우 집계 함수(Window Aggregate Function)로 사용할 수 있다. 새로운 기하 함수 (Regression Function - VAR_SAMP, VAR_POP, STDDEV_SAMP, STDDEV_POP 등등) 들을 사용할 수 있다. 좀 더 많은 Analytic 함수는 Oracle Manual 을 참고하기 바란다.

  **<Window Aggregate Functions>**

  **SUM, AVG, MIN, MAX, STDDEV, VARIANCE, COUNT, FIRST_VALUE, LAST_VALUE**

  **<Syntax>**

```
{Window Aggregate Families} ({<value expression1>|*})
OVER ( [PARTITION BY <value expression2>[,...]]
        ORDER BY <value expression3> [collate clause>]
        [ASC|DESC] [NULLS FIRST|NULLS LAST] [,...]
        ROWS|RANGE {{UNBOUNDED PRECEDING|<value expression4> PRECEDING}
        |BETWEEN UNBOUNDED PRECEDING|<value expression4> PRECEDING}
        AND{CURRENT ROW | <value expression4> FOLLOWING}} )
```

**<Reporting Aggregate Functions>**

**SUM, AVG, MAX, MIN, COUNT, STDDEV, VARIANCE**

**<Syntax>**
**{*Reporting Aggregate Families*} ([ALL|DISTINCT] {<value expression1>|*})**
**OVER ([PARTITION BY <value expression2>[,...]])**

**① Analytic Function**

<value expression1> 에는 하나 이상의 컬럼 또는 적합한 표현식이 사용될 수 있다. Analytic Function 의 Argument 는 0 에서 3 개까지 사용 가능하고 Asterisk(*)는 COUNT(*)에서만 허용되며 DISTINCT 는 해당 집계 함수가 허용할 때만 지원된다.

**② OVER analytic_clause**

해당 함수가 쿼리 결과 집합에 대해 적용되라는 지시어로써 FROM, WHERE, GROUP BY 와 HAVING 구 이후에 계산되어진다. SELECT 구 또는 ORDER BY 구에 Analytic Function 을 사용할 수 있다.

**ⓐ PARTITION BY 구**
<value expression2> 에는 하나 이상의 컬럼 또는 적합한 표현식이 사용될 수 있고 하나 이상의 컬럼 또는 표현식에 의한 그룹으로 쿼리의 결과를 파티션한다. 이 구가 생략되면 단일 그룹처럼 쿼리 결과 집합이 처리된다.

**ⓑ ORDER BY 구**
<value expression3> 에는 하나 이상의 컬럼 또는 적합한 표현식이 사용될 수 있고 하나 이상의 컬럼 또는 표현식을 기준으로 파티션 내의 데이터를 정렬한다. 표현식은 컬럼의 별칭 또는 위치를 나타내는 숫자를 사용할 수 없다.

**ⓒ Windowing 구**

**Window Size 의 정의**

⊙ Physical Window : Physical Window Size 는 rows 로 환산하여 표현한다.

⊙ Logical Window
- Time Interval : Logical Window Size 는 Time Interval 로 환산하여 표현한다.
- Value Range : Logical Window Size 는 정렬된 순서에서 Current Value 와 이전 value 들과의 차로 환산하여 표현한다.

**Window Size 정의에 사용되는 Keywords**

**ROWS** : 물리적인 ROW 단위로 WINDOW 를 지정하는 것을 나타냄.

Window size 를 ROWS 로 정의할 때, <value expression>은 physical offset(the number of rows)을 나타낸다. 그러므로, <value expression>은 반드시 positive numeric value 를 나타내야 한다.
(만약, 음수일 때는 'out of range'라는 error message 를 내고, 소수일 때는 truncate 를 시킨 정수 값을 기준으로 연산한다. )

offset 의 기준은 ORDER BY 절에 표현된 <value expression>을 근거로 하기 때문에 ROWS 로 window size 조정을 하려할 때는 반드시 ORDER BY 절을 표현해야 한다.

**RANGE** : 논리적인 상대번지로 WINDOW 를 지정하는 것을 나타냄.

Window size 를 RANGE 로 정의할 때, <value expression>은 logical offset(numeric 이나 date interval 에 근거한 offset)을 나타낸다. 또한, ROWS 의 경우와 마찬가지로 <value expression>은 반드시 positive numeric value 를 나타내야 한다.

RANGE 로 window size 를 조정하려 할 때도 ROWS 와 마찬가지로 ORDER BY 절을 반드시 표현해야 함은 물론, 반드시 하나의 <value expression>만 표현해야 한다.

RANGE 에 표현되는 <value expression>이 numeric value 일 때, ORDER BY 절의 <value expression>은 반드시 NUMBER 나 DATE TYPE 이어야 한다.

RANGE 에 표현되는 <value expression>이 interval value 일 때, ORDER BY 절의 <value expression>은 반드시 DATE TYPE 이어야 한다.

**CURRENT ROW** - 윈도우의 시작 위치 또는 마지막 위치가 현재 로우임을 지시하는 예약어
**UNBOUNDED PRECEDING** - 윈도우의 시작 위치가 Partition 의 첫 번째 로우임을 지시하는 예약어
**UNBOUNDED FOLLOWING** - 윈도우의 마지막 위치가 Partition 의 마지막 로우임을 지시하는 예약어

**Windowing Clause 에 정의되는 Sliding Window 의 Default Size**

order by 절이 없을 때
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

order by 절이 있을 때
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

- **Windowing 구의 분류 및 의미**

각 윈도우의 크기는 해당 파티션의 크기를 넘을 수 없고, 윈도우 구(Windowing_Clause)에 따라 해당 파티션 내에서 윈도우의 크기가 유기적으로 결정된다. 여기서 Physical Window 와 Logical Window(Time Interval, Value Range)에 의한 분류 및 윈도우 구에 의해 유기적인 크기가 정해지는 Cumulative, Moving, Centered 형태의 분류에 따른 결과의 차이에 대해 살펴볼 것이다. 끝으로 Logical Window 의 경우 PARTITION BY 구와 ORDER BY 구의 표현식의 결합에 의해 로우가 정렬되어 질 때 정렬이 Unique 한 정렬인지 Non Unique 한 정렬인지에 따라 결과가 어떻게 다른지를 살펴볼 것이다.

<주 1> 아래의 도표는 위의 내용들을 설명하기 위해 사용되는 데이터임.

| 고객번호 | 판매일자 | 판매금액 |
|---|---|---|
| 100 | 20020301 | 1000 |
| 100 | 20020302 | 130 |
| 100 | 20020303 | 1500 |
| 100 | 20020304 | 900 |
| 100 | 20020304 | 300 |
| 100 | 20020305 | 2300 |
| 200 | 20020301 | 500 |
| 200 | 20020302 | 250 |
| 200 | 20020303 | 1000 |
| 200 | 20020304 | 1500 |
| 200 | 20020305 | 3500 |
| 200 | 20020305 | 200 |

### ① Cumulative Aggregate Function

각 파티션의 시작 위치가 윈도우의 시작 위치가 되고 현재 로우의 물리적 또는 논리적 위치가 윈도우의 종료 위치가 되어 여기에 해당 집계함수를 적용하는 형태.

```
SELECT CUSTCODE, SALEDATE, SALE_AMT,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) ACC_AMT1,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE
            RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) ACC_AMT2,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY
                                TO_DATE(SALEDATE,'YYYYMMDD')
            RANGE BETWEEN UNBOUNDED PRECEDING AND INTERVAL '0' DAY
                                FOLLOWING) ACC_AMT3,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE) ACC_AMT4
FROM SALE_TBL;
```

위의 쿼리문에서 붉은색의 글자가 Windowing_Clause 를 의미하고 ACC_AMT1 은 Physical 윈도우에 의해, ACC_AMT2 는 Logical 윈도우중에서 Value Range 에 의해, ACC_AMT3 은 Logical 윈도우 중에서 Time Interval 에 의해, ACC_AMT4 는 windowing_clause 를 생략했지만 ACC_AMT2 와 같은 의미를 가진다.

| 고객번호 | 판매일자 | 판매금액 | acc_amt1 | acc_amt2 | acc_amt3 | acc_amt4 |
|---|---|---|---|---|---|---|
| 100 | 20020301 | 1000 | 1000 | 1000 | 1000 | 1000 |
| 100 | 20020302 | 130 | 1130 | 1130 | 1130 | 1130 |
| 100 | 20020303 | 1500 | 2630 | 2630 | 2630 | 2630 |
| 100 | 20020304 | 900 | **3530** | **3830** | **3830** | **3830** |
| 100 | 20020304 | 300 | 3830 | 3830 | 3830 | 3830 |
| 100 | 20020305 | 2300 | 6130 | 6130 | 6130 | 6130 |
| 200 | 20020301 | 500 | 500 | 500 | 500 | 500 |
| 200 | 20020302 | 250 | 750 | 750 | 750 | 750 |
| 200 | 20020303 | 1000 | 1750 | 1750 | 1750 | 1750 |
| 200 | 20020304 | 1500 | 3250 | 3250 | 3250 | 3250 |
| 200 | 20020305 | 3500 | **6750** | **6950** | **6950** | **6950** |
| 200 | 20020305 | 200 | 6950 | 6950 | 6950 | 6950 |

위의 도표에서 고객번호가 '100'이고 판매일자가 '20020304', 판매금액이 900 원인 로우를 기준으로 보면 검은 화살표는 파티션, 빨간 화살표는 물리적인 윈도우 크기(Physical Window Size), 파란 화살표는 논리적인 윈도우 크기(Logical Window Size)를 의미한다. 여기서 주의해서 봐야 할 것은 논리적인 윈도우의 크기가 물리적인 윈도우 크기보다 한 로우 더 내려간 것을 볼 수 있을 것이다. 이것은 데이터가 고객번호, 판매일자 순으로 정렬되므로 고객번호 '100', 판매일자 '20020304' 기준으로 보면 판매액이 900 원 300 원이 발생했지만 정렬순위가 같으므로 논리적으로는 1200 원으로 처리된다.

### ② Moving Aggregate Function

각 파티션내의 현재 로우의 물리적 또는 논리적 위치가 윈도우의 종료 위치가 되고 이 종료 위치를 기준으로 OFFSET 을 적용하여 시작위치가 결정되고 여기에 해당 집계함수를 적용하는 형태.

```
SELECT CUSTCODE, SALEDATE, SALE_AMT,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE
            ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) ACC_AMT1,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY
                                                TO_DATE(SALEDATE,'YYYYMMDD')
            RANGE BETWEEN 1 PRECEDING AND CURRENT ROW) ACC_AMT2,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY
                                                TO_DATE(SALEDATE,'YYYYMMDD')
            RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND CURRENT ROW) ACC_AMT3
FROM SALE_TBL;
```

위의 쿼리문에서 붉은색의 글자가 Windowing_Clause 를 의미하고 ACC_AMT1 은 Physical 윈도우에 의해, ACC_AMT2 는 Logical 윈도우중에서 Value Range 에 의해, ACC_AMT3 은 Logical 윈도우 중에서 Time Interval 에 대한 의미를 가진다.

| 고객번호 | 판매일자 | 판매금액 | acc_amt1 | acc_amt2 | acc_amt3 | |
|---|---|---|---|---|---|---|
| 100 | 20020301 | 1000 | 1000 | 1000 | 1000 | |
| 100 | 20020302 | 130 | 1130 | 1130 | 1130 | |
| 100 | 20020303 | 1500 | 1630 | 1630 | 1630 | |
| 100 | 20020304 | 900 | 2400 | 2700 | 2700 | |
| 100 | 20020304 | 300 | 1200 | 2700 | 2700 | |
| 100 | 20020305 | 2300 | 2600 | 2600 | 2600 | |
| 200 | 20020301 | 500 | 500 | 500 | 500 | |
| 200 | 20020302 | 250 | 750 | 750 | 750 | |
| 200 | 20020303 | 1000 | 1250 | 1250 | 1250 | |
| 200 | 20020304 | 1500 | 2500 | 2500 | 2500 | |
| 200 | 20020305 | 3500 | 5000 | 5200 | 5200 | |
| 200 | 20020305 | 200 | 3700 | 5200 | 5200 | |

위의 도표에서 고객번호가 '100'이고 판매일자가 '20020304', 판매금액이 900 원인 로우를 기준으로 보면 검은 화살표는 파티션, 빨간 화살표는 물리적인 윈도우 크기(Physical Window Size), 파란 화살표는 논리적인 윈도우 크기(Logical Window Size)를 의미한다. 여기서 주의해서 봐야 할 것은 논리적인 윈도우의 크기가 물리적인 윈도우 크기보다 한 로우 더 내려간 것을 볼 수 있을 것이다. 이것은 데이터가 고객번호, 판매일자 순으로 정렬되므로 고객번호 '100', 판매일자 '20020304' 기준으로 보면 판매액이 900 원 300 원이 발생했지만 정렬순위가 같으므로 논리적으로는 1200 원으로 처리된다.

### ③ Centered Aggregate Function

각 파티션내의 윈도우의 크기가 현재 로우의 물리적 또는 논리적 위치를 기준으로 주어진 오프셋(Offset)을 적용하여 윈도우의 시작위치와 종료위치가 결정되고 여기에 해당 집계함수를 적용하는 형태.

```
SELECT CUSTCODE, SALEDATE, SALE_AMT,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE
                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) ACC_AMT1,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY
                                        TO_DATE(SALEDATE,'YYYYMMDD')
                RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) ACC_AMT2,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY
                                        TO_DATE(SALEDATE,'YYYYMMDD')
                RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
                                INTERVAL '1' DAY FOLLOWING) ACC_AMT3
FROM SALE_TBL;
```

위의 쿼리문에서 붉은색의 글자가 Windowing_Clause 를 의미하고 ACC_AMT1 은 Physical 윈도우에 의해, ACC_AMT2 는 Logical 윈도우중에서 Value Range 에 의해, ACC_AMT3 은 Logical 윈도우 중에서 Time Interval 에 대한 의미를 가진다.

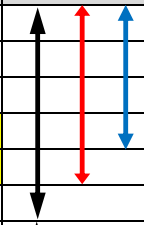
| 고객번호 | 판매일자 | 판매금액 | acc_amt1 | acc_amt2 | acc_amt3 | |
|---|---|---|---|---|---|---|
| 100 | 20020301 | 1000 | 1130 | 1130 | 1130 | |
| 100 | 20020302 | 130 | 2630 | 2630 | 2630 | |
| 100 | 20020303 | 1500 | 2530 | 2830 | 2830 | |
| 100 | 20020304 | 900 | 2700 | 5000 | 5000 | |
| 100 | 20020304 | 300 | 3500 | 5000 | 5000 | |
| 100 | 20020305 | 2300 | 2600 | 3500 | 3500 | |
| 200 | 20020301 | 500 | 750 | 750 | 750 | |
| 200 | 20020302 | 250 | 1750 | 1750 | 1750 | |
| 200 | 20020303 | 1000 | 2750 | 2750 | 2750 | |
| 200 | 20020304 | 1500 | 6000 | 6200 | 6200 | |
| 200 | 20020305 | 3500 | 5200 | 5200 | 5200 | |
| 200 | 20020305 | 200 | 3700 | 5200 | 5200 | |

위의 도표에서 고객번호가 '100'이고 판매일자가 '20020304', 판매금액이 900 원인 로우를 기준으로 보면 검은 화살표는 파티션, 빨간 화살표는 물리적인 윈도우 크기(Physical Window Size), 파란 화살표는 논리적인 윈도우 크기(Logical Window Size)를 의미한다. 여기서 주의해서 봐야 할 것은 논리적인 윈도우의 크기가 물리적인 윈도우 크기보다 한 로우 더 내려간 것을 볼 수 있을 것이다. 이것은 데이터가 고객번호, 판매일자 순으로 정렬되므로 고객번호 '100', 판매일자 '20020304' 기준으로 보면 판매액이 900 원 300 원이 발생했지만 정렬순위가 같으므로 논리적으로는 1200 원으로 처리된다.

**④ Window Size 가 Logical Offset 으로 결정될 경우 Ordering 이 Unique 할때와 Non Unique 할 때의 차이점**

```
SELECT CUSTCODE, SALEDATE, SALE_AMT,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE
                RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) ACC_AMT1,
        SUM(SALE_AMT) OVER (PARTITION BY CUSTCODE ORDER BY SALEDATE, SALE_AMT
                RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) ACC_AMT2
FROM SALE_TBL;
```

위의 쿼리문에서 푸른색의 글자가 데이터 정렬에 기준이 되는 PARTITION BY 와 ORDER BY 구이고 붉은색의 글자가 Windowing_Clause 를 의미하고 ACC_AMT1 과 ACC_AMT2 은 둘 다 Value Range 에 의한 논리적 윈도우의 결과를 반환하지만, ORDER BY 의 표현식을 다르게 하였다. ACC_AMT1 의 데이터 정렬은 CUSTCODE, SALEDATE 기준으로 수행되고 데이터와 비교해 보면 정렬의 순서가 같은 데이터 들이 있다. 그러나 ACC_AMT2 는 데이터 정렬이 CUSTCODE, SALEDATE, SALE_AMT 기준으로 수행되고 데이터와 비교해 보면 정렬 순서가 Unique 하게 됨을 알 수 있다.

| 고객번호 | 판매일자 | 판매금액 | acc_amt1 | acc_amt2 | |
|---|---|---|---|---|---|
| 100 | 20020301 | 1000 | 1000 | 1000 |  |
| 100 | 20020302 | 130 | 1130 | 1130 | |
| 100 | 20020303 | 1500 | 2630 | 2630 | |
| 100 | 20020304 | 300 | **3830** | **2930** | |
| 100 | 20020304 | 900 | 3830 | 3830 | |
| 100 | 20020305 | 2300 | 6130 | 6130 | |
| 200 | 20020301 | 500 | 500 | 500 | |
| 200 | 20020302 | 250 | 750 | 750 | |
| 200 | 20020303 | 1000 | 1750 | 1750 | |
| 200 | 20020304 | 1500 | 3250 | 3250 | |
| 200 | 20020305 | 200 | **6950** | **3450** | |
| 200 | 20020305 | 3500 | 6950 | 6950 | |

위의 도표에서 고객번호가 '100'이고 판매일자가 '20020304', 판매금액이 300 원인 로우를 기준으로 보면 검은 화살표는 파티션, 빨간 화살표는 ACC_AMT1 의 논리적인 윈도우 크기, 파란 화살표는 ACC_AMT2 의 논리적인 윈도우 크기를 (Logical Window Size)를 의미한다. 여기서 주의해서 봐야 할 것은 ACC_AMT1 의 논리적인 윈도우의 크기가 ACC_AMT2 의 논리적인 윈도우 크기보다 한 로우 더 내려간 것을 볼 수 있을 것이다. 이것은 ACC_AMT1 의 경우 데이터가 고객번호, 판매일자 순으로 정렬되므로 고객번호 '100', 판매일자 '20020304' 기준으로 보면 판매액이 300 원 900 원이 발생했지만 정렬순위가 같으므로 논리적으로는 1200 원으로 처리된다. 그러나 ACC_AMT2 의 경우 데이터가 고객번호, 판매일자, 판매금액 순으로 정렬되므로 고객번호 '100', 판매일자 '20020304' 이고 판매액이 300 원인 경우가 판매액이 900 원인 경우 보다 정렬우선 순위가 앞서므로 각각의 로우가 별개로 처리됨을 알 수 있다. 위의 도표의 결과에서 알 수 있듯이 논리적 윈도우 구를 지정하게 되면 정렬의 기준이 되는 PARTITION BY 와 ORDER BY 구에 의한 정렬의 기준이 Unique 한 경우와 Non Unique 한 경우 처리 결과가 다름을 알 수 있다.

- **Window Aggregate 및 Reporting Aggregate Function 의 활용 사례**

**① Window Aggregate Family – SUM( )**

다음과 같이 데이터가 발생하였을 때 일별 판매액 누계를 구하는 쿼리문을 Analytic Function 을 사용하지 않은 경우와 사용한 경우를 살펴보자.

<주 2> 도표에서 회색으로 보이는 부분은 원래 데이터이고 노란색으로 보이는 부분은 원하는 결과임.

| 판매일자 | 판매액 | 판매누계액 |
|---|---|---|
| 20020301 | 1000 | 1000 |
| 20020302 | 130 | 1130 |
| 20020303 | 1500 | 2630 |
| 20020304 | 900 | 3530 |
| 20020305 | 2300 | 5830 |

ⓐ Analytic Function 을 사용하지 않은 경우

먼저, 일별 판매액 누계를 구하고자 하면 다음과 같이
2002 년 03 월 01 일 기준 : 2002 년 03 월 01 일 판매액
2002 년 03 월 02 일 기준 : 2002 년 03 월 01 일 판매액 + 2002 년 03 월 02 일 판매액
2002 년 03 월 03 일 기준 : 2002 년 03 월 01 일 판매액 + 2002 년 03 월 02 일 판매액 +
2002 년 03 월 03 일 판매액 。。。。。。
형태의 데이터가 필요하다.
이런 형태의 중간집합을 만들고자 할 때 데이터 복제(COPY_T 이용-대용량 데이타베이스 2 권 참조)를 통하여 필요한 집합을 만들고 이 집합에 대하여 GROUPING 을 하여 최종 결과를 도출한다. 그러나 이 경우는 데이터 복제가 발생하고 그룹핑에 대한 비효율이 존재한다.

즉, 아래와 같은 쿼리문을 작성하여 일자별 판매액 누계를 구할 수 있다.

```
SELECT MIN(DECODE(T1.NO, V1.RCNT ,V1.SALEDATE)) SALEDATE,
       SUM(V1.SALE_AMT) SALE_AMT
 FROM (SELECT SALEDATE * –1 || '' SALEDATE, SALE_AMT, ROWNUM RCNT
         FROM (SELECT SALEDATE * -1 SALEDATE, SUM(SALE_AMT) SALE_AMT
                 FROM SALE_TBL
                GROUP BY SALEDATE * –1
               )
       ) V1, COPY_T T1
 WHERE V1.RCNT >= T1.NO
 GROUP BY T1.NO
 ORDER BY SALEDATE;
```

ⓑ Analytic Function 을 사용한 경우

```
SELECT SALEDATE,
       SUM(SALE_AMT) OVER (ORDER BY SALEDATE) SALE_AMT
 FROM SALE_TBL;
```

**② Window Aggregate Family – MAX( )**

다음과 같이 데이터가 발생하였을 때 상태가 '신규', '명의변경', '기기변경'인 경우의 시작일과 상태코드를 상태변경일과 상태변경코드로 관리하고 그 이외의 상태는 '신규', '명의변경', '기기변경'의 상태 변경일과 상태변경코드로 상속을 받는 형태의 결과를 구하는 쿼리문을 Analytic Function 을 사용하지 않은 경우와 사용한 경우를 살펴보자.

<주 3> 도표에서 회색으로 보이는 부분은 원래 데이터이고 노란색으로 보이는 부분은 원하는 결과임.

| 고객 번호 | 이름 | 시작일 | 종료일 | 상태 코드 | 상태명 | 상태 변경일 | 상태변경 코드 |
|---|---|---|---|---|---|---|---|
| 100 | 홍길동 | 19980102 | 19990302 | 30 | 개통 | 19980102 | 30 |
| 100 | 홍길동 | 19990302 | 19990421 | 70 | 정지 | 19980102 | 30 |
| 100 | 홍길동 | 19990421 | 20000823 | 50 | 복구 | 19980102 | 30 |
| 100 | 차인표 | 20000823 | 20011002 | 10 | 명의변경 | 20000823 | 10 |
| 100 | 홍길동 | 20001002 | 99991231 | 10 | 명의변경 | 20001002 | 10 |
| 200 | 이순신 | 20000401 | 20000523 | 30 | 개통 | 20000401 | 30 |
| 200 | 이순신 | 20000523 | 20000823 | 70 | 정지 | 20000401 | 30 |
| 200 | 이순신 | 20000823 | 20010418 | 50 | 복구 | 20000401 | 30 |
| 200 | 김유신 | 20010418 | 20010827 | 10 | 명의변경 | 20010418 | 10 |
| 200 | 김유신 | 20010827 | 20010911 | 70 | 정지 | 20010418 | 10 |
| 200 | 김유신 | 20010911 | 20011020 | 50 | 복구 | 20010418 | 10 |
| 200 | 김유신 | 20011020 | 20011215 | 20 | 기기변경 | 20011020 | 20 |
| 200 | 이순신 | 20011215 | 99991231 | 10 | 명의변경 | 20011215 | 10 |

ⓐ Analytic Function 을 사용하지 않은 경우

'신규', '명의변경','기기변경'인 경우만 선택하여 다음과 같이 선분이력을 먼저 만든다.

| 고객번호 | 시작일 | 종료일 | 상태코드 |
|---|---|---|---|
| 100 | 19980102 | 20000823 | 30 |
| 100 | 20000823 | 20001002 | 10 |
| 100 | 20001002 | 99991231 | 10 |
| 200 | 20000401 | 20010418 | 30 |
| 200 | 20010418 | 20011020 | 10 |
| 200 | 20011020 | 20011215 | 20 |
| 200 | 20011215 | 99991231 | 10 |

그런 후 아래의 쿼리문과 같이 위의 도표와 같이 만들어진 중간 집합으로 원래의 테이블과 BETWEEN JOIN 을 하여 상태 변경일을 상속 받는다. 아래 쿼리문에서 알 수 있듯이 같은 테이블을 여러번 Access 하고 그룹핑을 한 후 자기 자신과 조인을 해야 하는 비효율이 발생한다.

<주 4> 아래 붉은색으로 칠해진 쿼리가 위의 도표처럼 해당 상태에 대한 선분이력을 만드는 쿼리임.

```
SELECT T1.CONTNO, T1.NAME, T1.SDATE, T1.EDATE, T1.STATE_CODE, T1.STATE_NAME,
          V3.SDATE CHG_DATE, V3.STATE_CODE CHG_CODE
  FROM (SELECT /*+ ORDERED USE_MERGE(V1 V2) */
                   V1.CONTNO, V1.SDATE, V1.STATE_CODE,
                   DECODE(V2.SDATE, NULL, '99991231',
                              to_char(to_date(V2.SDATE, 'yyyymmdd')-1, 'yyyymmdd')) EDATE
             FROM (SELECT CONTNO, SDATE, STATE_CODE, ROWNUM RCNT
                     FROM (
                              SELECT CONTNO, SDATE, STATE_CODE
                               FROM CONT_TBL
                              WHERE STATE_CODE IN ('30','20','10')
                              GROUP BY CONTNO, SDATE, STATE_CODE
                             )
                   ) V1,
                   (
                    SELECT CONTNO, SDATE, STATE_CODE, ROWNUM-1 RCNT
                     FROM (
                              SELECT CONTNO, SDATE, STATE_CODE
                               FROM CONT_TBL
                               WHERE STATE_CODE IN ('30','20','10')
                               GROUP BY CONTNO, SDATE, STATE_CODE
                              )
                   ) V2
             WHERE V2.CONTNO(+) = V1.CONTNO
               AND V2.RCNT(+)     = V1.RCNT
           ) V3, CONT_TBL T1
 WHERE T1.CONTNO = V3.CONTNO
   AND T1.EDATE > V3.SDATE
   AND T1.SDATE < V3.EDATE;
```

ⓑ Analytic Function 을 사용한 경우

Analytic Function 을 활용하면 간단하게 해결할 수 있지만 약간의 데이터 조작이 필요하다. 상태 변경일의 경우는 변경일이 계속 커지는 경우이기 때문에 '신규', '명의변경','기기변경'의 경우에만 시작일을 그대로 보존하고 나머지 상태는 NULL 로 치환한 후 MAX()함수를 가지고 Analytic Function 을 적용하면 원하는 결과를 얻을 수 있지만, 상태 코드의 경우는 크기가 서로 혼재되어 있으므로 약간의 조작이 더 필요하다. 후자의 경우는 ROW_NUMBER()를 이용하여 MAX()에 대한 크기를 보정해 주어야 한다. 이해를 돕기 위해 아래 데이터에 대한 도표를 참고 하기 바란다.

| 고객 번호 | 이름 | 시작일 | 종료일 | 상태 코드 | 상태명 | 상태 변경일 | 상태변경 코드 | T_RANK |
|---|---|---|---|---|---|---|---|---|
| 100 | 홍길동 | 19980102 | 19990302 | 30 | 개통 | 19980102 | 30 | 1 |
| 100 | 홍길동 | 19990302 | 19990421 | 70 | 정지 | | | 2 |
| 100 | 홍길동 | 19990421 | 20000823 | 50 | 복구 | | | 3 |
| 100 | 차인표 | 20000823 | 20011002 | 10 | 명의변경 | 20000823 | 10 | 4 |
| 100 | 홍길동 | 20001002 | 99991231 | 10 | 명의변경 | 20001002 | 10 | 5 |
| 200 | 이순신 | 20000401 | 20000523 | 30 | 개통 | 20000401 | 30 | 1 |
| 200 | 이순신 | 20000523 | 20000823 | 70 | 정지 | | | 2 |
| 200 | 이순신 | 20000823 | 20010418 | 50 | 복구 | | | 3 |
| 200 | 김유신 | 20010418 | 20010827 | 10 | 명의변경 | 20010418 | 10 | 4 |
| 200 | 김유신 | 20010827 | 20010911 | 70 | 정지 | | | 5 |
| 200 | 김유신 | 20010911 | 20011020 | 50 | 복구 | | | 6 |
| 200 | 김유신 | 20011020 | 20011215 | 20 | 기기변경 | 20011020 | 20 | 7 |
| 200 | 이순신 | 20011215 | 99991231 | 10 | 명의변경 | 20011215 | 10 | 8 |

<주 5> 아래 파란색으로 칠해진 쿼리가 위의 도표와 같은 결과를 만드는 부분이고
자주색으로 칠해진 쿼리가 상태변경일 및 상태코드를 상속받을 수 있도록 보정하는 쿼리임.

```
SELECT CONTNO, NAME, SDATE, EDATE, STATE_CODE, STATE_NAME,
       MAX(CHG_DATE) OVER (PARTITION BY CONTNO ORDER BY SDATE) CHG_DATE,
       SUBSTR(MAX(DECODE(CHG_CODE,NULL,NULL,
                         TO_CHAR(T_RANK,'B99999999')||CHG_CODE))
              OVER (PARTITION BY CONTNO ORDER BY SDATE),-2) CHG_CODE
FROM (
       SELECT CONTNO, NAME, SDATE, EDATE, STATE_CODE, STATE_NAME,
              DECODE(STATE_CODE,'10',SDATE,'20',SDATE,'30',SDATE) CHG_DATE,
              DECODE(STATE_CODE,'10',STATE_CODE,'20',STATE_CODE,'30',STATE_CODE)
                                                          CHG_CODE,
              ROW_NUMBER( )
                     OVER (PARTITION BY CONTNO ORDER BY SDATE) T_RANK
       FROM CONT_TBL
       );
```

다음은 아래의 EMP 테이블을 이용하여 Reporting Aggregate Function 의 활용사례에 대해 설명할 것이다.

| EMPNO | ENAME | JOB | MGR | HIREDATE | SAL | COMM | DEPTNO |
|---|---|---|---|---|---|---|---|
| 7369 | SMITH | CLERK | 7902 | 1980-12-17 | 800 | | 20 |
| 7499 | ALLEN | SALESMAN | 7698 | 1981-02-20 | 1600 | 300 | 30 |
| 7521 | WARD | SALESMAN | 7698 | 1981-02-22 | 1250 | 500 | 30 |
| 7566 | JONES | MANAGER | 7839 | 1981-04-02 | 2975 | | 20 |
| 7654 | MARTIN | SALESMAN | 7698 | 1981-09-28 | 1250 | 1400 | 30 |
| 7698 | BLAKE | MANAGER | 7839 | 1981-05-01 | 2850 | | 30 |
| 7782 | CLARK | MANAGER | 7839 | 1981-06-09 | 2450 | | 10 |
| 7788 | SCOTT | ANALYST | 7566 | 1982-12-09 | 3000 | | 20 |
| 7839 | KING | PRESIDENT | | 1981-11-17 | 5000 | | 10 |
| 7844 | TURNER | SALESMAN | 7698 | 1981-09-08 | 1500 | 0 | 30 |
| 7876 | ADAMS | CLERK | 7788 | 1983-01-12 | 1100 | | 20 |
| 7900 | JAMES | CLERK | 7698 | 1981-12-03 | 950 | | 30 |
| 7902 | FORD | ANALYST | 7566 | 1981-12-03 | 3000 | | 20 |
| 7934 | MILLER | CLERK | 7782 | 1982-01-23 | 1300 | | 10 |

**③ Reporting Aggregate Family - SUM( )**

위의 도표와 같이 데이터가 발생하였을 때 각각의 부서별로, 총 급여액이 가장 많은 직무를 알아보는 경우에 대해 Analytic Function 을 사용한 경우와 사용하지 않은 경우를 살펴보자.

<주 6> 아래의 도표가 원하는 결과임.

| DEPTNO | JOB | SUM_SAL | MAX_SUM_SAL |
|---|---|---|---|
| 10 | PRESIDENT | 5000 | 5000 |
| 20 | ANALYST | 6000 | 6000 |
| 30 | SALESMAN | 5600 | 5600 |

ⓐ Analytic Function 을 사용하지 않은 경우

아래 쿼리문에서 알 수 있듯이 같은 테이블을 두 번 Access 하고 서로 다른 집합 레벨로 인하여 여러 번의 그룹핑을 한 후 서로 조인을 해야 하는 비효율이 발생한다.

```
SELECT V2.DEPTNO, V2.JOB, V2.SUM_SAL, V1.MAX_SUM_SAL
 FROM (
         SELECT DEPTNO, MAX(SUM_SAL) MAX_SUM_SAL
          FROM (
                  SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL
                   FROM EMP
                  GROUP BY DEPTNO, JOB
                  )
         GROUP BY DEPTNO
         ) V1,
         (
          SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL
           FROM EMP
           GROUP BY DEPTNO, JOB
         ) V2
WHERE V2.DEPTNO = V1.DEPTNO
  AND V2.SUM_SAL = V1.MAX_SUM_SAL;
```

ⓑ Analytic Function 을 사용한 경우

```
SELECT DEPTNO, JOB, SUM_SAL, MAX_SUM_SAL
 FROM (
         SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL,
                 MAX(SUM(SAL)) OVER (PARTITION BY DEPTNO) MAX_SUM_SAL
          FROM EMP
         GROUP BY DEPTNO, JOB
        )
WHERE SUM_SAL = MAX_SUM_SAL;
```

**④ Reporting Aggregate Family - RATIO_TO_REPORT( )**

각각의 부서 총 급여액에 대한 부서별, 직무 별 총 판매액의 비율을 알고자 할 경우 Analytic Function 을 사용한 경우와 사용하지 않은 경우를 살펴보자. 참고적으로 RATIO_TO_REPORT( ) 함수는 윈도우내의 합계에 대한 비율을 계산하는 함수이다.

<주 7> 아래의 도표가 원하는 결과임.

| DEPTNO | JOB | SUM_SAL | SUM_TOTAL | RATIO_TO_SUM |
|---|---|---|---|---|
| 10 | CLERK | 1300 | 8750 | 0.148571428571429 |
| 10 | MANAGER | 2450 | 8750 | 0.28 |
| 10 | PRESIDENT | 5000 | 8750 | 0.571428571428571 |
| 20 | ANALYST | 6000 | 10875 | 0.551724137931034 |
| 20 | CLERK | 1900 | 10875 | 0.174712643678161 |
| 20 | MANAGER | 2975 | 10875 | 0.273563218390805 |
| 30 | CLERK | 950 | 9400 | 0.101063829787234 |
| 30 | MANAGER | 2850 | 9400 | 0.303191489361702 |
| 30 | SALESMAN | 5600 | 9400 | 0.595744680851064 |

ⓐ Analytic Function 을 사용하지 않은 경우

아래 쿼리문에서 알 수 있듯이 같은 테이블을 두 번 Access 하고 서로 다른 집합 레벨로 인하여 두 번의 그룹핑을 한 후 서로 조인을 해야 하는 비효율이 발생한다.

```
SELECT V2.DEPTNO, V2.JOB, V2.SUM_SAL, V1.SUM_TOTAL,
       V2.SUM_SAL/V1.SUM_TOTAL RATIO_TO_SUM
 FROM (
        SELECT DEPTNO, SUM(SAL) SUM_TOTAL
          FROM EMP
         GROUP BY DEPTNO
       ) V1,
       (
        SELECT DEPTNO, JOB, SUM(SAL) AS SUM_SAL
          FROM EMP
         GROUP BY DEPTNO, JOB
       ) V2
 WHERE V2.DEPTNO = V1.DEPTNO;
```

ⓑ Analytic Function 을 사용한 경우

```
SELECT DEPTNO, JOB, SUM(SAL) SUM_SAL,
       SUM(SUM(SAL)) OVER (PARTITION BY DEPTNO) SUM_TOTAL,
       RATIO_TO_REPORT(SUM(SAL)) OVER (PARTITION BY DEPTNO) RATIO_TO_SUM
 FROM EMP
GROUP BY DEPTNO, JOB;
```

### 4) Lead/Lag Family 의 소개 및 활용 사례

- **Lead/Lag Family 의 특징**

  Lead/Lag Family 는 특정 로우가 속한 파티션([엔코아 이창수 수석]제 1 회 Analytic Function 의 소개 글의 Analytic function 의 3 요소 중 Result Set Partitions 참조) 내에서 상대적 상하 위치에 있는 특정 로우의 컬럼 값을 참조하거나 상호 비교하고자 할 때 사용할 수 있는 function 들의 집합으로, 그 특징은 다음과 같다.

  - 오름차순 또는 내림차순으로 정렬된 파티션 내에서 상대적으로 상위 또는 하위에 위치하고 있는 특정 로우의 컬럼 값을 offset 지정에 의해 참조할 수 있다.

  - 파티션 내에서 참조할 로우가 없을 경우 지정한 값(default = NULL)으로 출력한다.

  - Order by 에 기술된 컬럼의 값이 NULL 인 경우 오름차순 또는 내림차순과 관계없이 순서상 가장 처음 또는 마지막으로 강제 처리 가능하다.

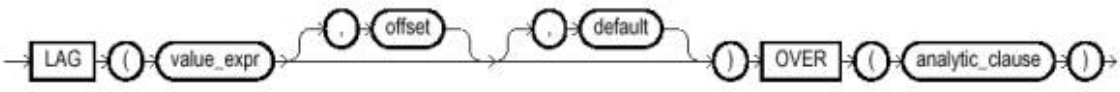
  **<Syntax>**

  ```
  {LAG|LEAD} (<value expression1>, [<offset> [, <default>]])
  OVER ([PARTITION BY <value expression2>[,...]]
        ORDER BY <value expression3> [collate clause>]
                 [ASC|DESC] [NULLS FIRST| NULLS LAST] [,...])
  ```

- **Lead/Lag Family 의 종류**

  **① Lag( )**

  Lag 함수는 파티션 내에서 offset 에 지정된 값(default = 1) 만큼 상대적으로 상위에 위치한 로우(오름차순의 경우 기준 로우의 정렬 컬럼 값보다 작은 값을 갖는 로우, 내림차순의 경우 기준 로우의 정렬 컬럼 값보다 큰 값을 갖는 로우)를 참조하기 위해 사용된다.

  ***offset***
  파티션 내에서 참조하고자 하는 로우를 지정하기 위한 대한 상대 주소이다. 지정 하지 않으면 1 이다.

  ***default***
  Offset 에 의해 참조되는 로우가 파티션 내에 없을 경우, 즉 파티션의 범위를 벗어나는 경우에 출력되는 값을 지정한다. 지정하지 않으면 null 이다.

  ***analytic_clause***
  파티션의 크기를 지정하며 파티션 내 로우들에 대한 정렬 순서를 결정한다.

다음 테이블은 신용카드의 마일리지 포인트에 대한 포인트 적립 또는 포인트 사용 이력을 표현한 것이다

| 카드번호 | 포인트적립사용일자 | 순번 | 적립사용포인트 |
|---|---|---|---|
| A1000876 | 20011010 | 1 | 120 |
| A1000876 | 20011006 | 1 | 100 |
| A1000876 | 20011006 | 2 | -80 |
| A1000876 | 20011030 | 1 | 500 |
| A2000231 | 20011009 | 1 | 300 |
| A2000231 | 20011101 | 1 | 400 |
| A2000231 | 20011205 | 1 | -500 |
| A2000231 | 20011205 | 2 | 300 |
| A2000231 | 20020110 | 1 | -350 |
| A3000567 | 20011010 | 1 | 600 |
| A3000567 | 20011001 | 2 | 200 |
| A3000567 | 20011101 | 1 | 100 |
| A3000567 | 20011010 | 3 | -300 |
| A3000567 | 20011010 | 2 | -100 |

위의 테이블의 각 로우에 대해 동일 카드번호의 종전 마일리지 포인트적립사용일자 및 적립사용포인트를 보고자 할 경우 다음과 같이 lag( )를 사용하여 간단히 그 결과를 볼 수 있다.

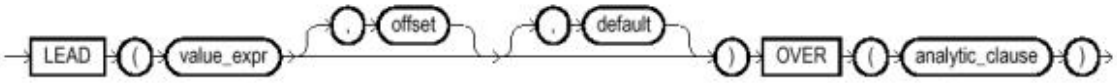
**SQL:**

```
SELECT 카드번호, 포인트적립사용일자, 순번, 적립사용포인트,
        lag(포인트적립사용일자) over (partition by 카드번호
            order by 포인트적립사용일자, 순번) as 종전포인트적립사용일자,
        lag(적립사용포인트) over (partition by 카드번호
            order by 포인트적립사용일자, 순번) as 적립사용포인트
FROM 마일리지테이블;
```

**결과:**

**② Lead( )**

Lead( )는 Lag( )와 유사한 함수로 offset 에 지정된 값(default = 1) 만큼 상대적으로 하위에 위치한 로우(오름차순의 경우 기준 로우의 정렬 컬럼 값보다 큰 값을 갖는 로우, 내림차순의 경우 기준 로우의 정렬 컬럼 값보다 작은 값을 갖는 로우)를 참조하기 위해 사용된다는 점만 다르다.




Lag( ) 예제와는 반대로 마일리지테이블의 각 로우에 대해 동일 카드번호의 바로 다음 적립 또는 사용한 일자와 포인트를 보고자 할 경우 다음과 같이 lead( )를 사용하여 그 결과를 볼 수 있다.

**SQL:**

```
SELECT 카드번호,
       포인트적립사용일자, 순번, 적립사용포인트,
        lead(포인트적립사용일자) over (partition by 카드번호
            order by 포인트적립사용일자, 순번) as 종전포인트적립사용일자,
        lead(적립사용포인트) over (partition by 카드번호
            order by 포인트적립사용일자, 순번) as 적립사용포인트
 FROM 마일리지테이블;
```

**결과:**


| 카드번호 | 포인트적립 사용일자 | 순번 | 적립사용 포인트 | 종전포인트 적립사용일자 | 종전적립 사용포인트 |
|---|---|---|---|---|---|
| A1000876 | 20011006 | 1 | 100 | 20011006 | -80 |
| A1000876 | 20011006 | 2 | -80 | 20011010 | 120 |
| A1000876 | 20011010 | 1 | 120 | 20011030 | 500 |
| A1000876 | 20011030 | 1 | 500 | NULL | NULL |
| A2000231 | 20011009 | 1 | 300 | 20011101 | 400 |
| A2000231 | 20011101 | 1 | 400 | 20011205 | -500 |
| A2000231 | 20011205 | 1 | -500 | 20011205 | 300 |
| A2000231 | 20011205 | 2 | 300 | 20020110 | -350 |
| A2000231 | 20020110 | 1 | -350 | NULL | NULL |
| A3000567 | 20011001 | 2 | 200 | 20011010 | 600 |
| A3000567 | 20011010 | 1 | 600 | 20011010 | -100 |
| A3000567 | 20011010 | 2 | -100 | 20011010 | -300 |
| A3000567 | 20011010 | 3 | -300 | 20011101 | 100 |
| A3000567 | 20011101 | 1 | 100 | NULL | NULL |

- **Lead/Lag Family 의 활용 사례**

**① 신용카드 마일리지 포인트 적립 또는 사용 이력의 선분화**

Lag/Lead family 사용 예에서 언급한 바 있는 신용카드 마일리지 포인트 적립 또는 사용에 관한 이력을 선분화 해보자. 아래의 마일리지테이블은 마일리지 포인트가 적립되거나 사용된 일자와 해당 포인트를 관리하고 있다.

대상 테이블:

| 카드번호 | 포인트적립사용일자 | 순번 | 적립사용포인트 |
|---|---|---|---|
| A1000876 | 20011010 | 1 | 120 |
| A1000876 | 20011006 | 1 | 100 |
| A1000876 | 20011006 | 2 | -80 |
| A1000876 | 20011030 | 1 | 500 |
| A2000231 | 20011009 | 1 | 300 |
| A2000231 | 20011101 | 1 | 400 |
| A2000231 | 20011205 | 1 | -500 |
| A2000231 | 20011205 | 2 | 300 |
| A2000231 | 20020110 | 1 | -350 |
| A3000567 | 20011010 | 1 | 600 |
| A3000567 | 20011001 | 2 | 200 |
| A3000567 | 20011101 | 1 | 100 |
| A3000567 | 20011010 | 3 | -300 |
| A3000567 | 20011010 | 2 | -100 |

일반적으로, 운영 시스템은 마일리지 포인트가 적립되거나 사용된 business event 를 중시하여 데이터화하기 때문에 event 가 발생한 일자(마일리지 포인트 적립 또는 사용일자)와 event 결과(적립 또는 사용된 마일리지 포인트)에만 관심을 갖는다. 그러나 이러한 데이터 모델은 신용카드별 특정 시점의 사용 가능한 마일리지 포인트를 조회하고자 하는 요구사항에 대응하기 어렵다. 이와 같은 요구사항에 쉽게 대처하기 위해서 카드번호, 유효시작일자, 유효종료일자, 사용가능포인트 컬럼으로 구성된 사용가능포인트테이블을 설계하고 기존 마일리지테이블로부터 데이터를 추출하여 로딩하는 SQL 을 Lead( )를 활용하여 작성해보자. 사용가능포인트테이블의 유효시작일자는 마일리지 포인트를 적립 또는 사용한 일자이며, 유효종료일자는 그 다음에 발생한 마일리지 포인트 적립 또는 사용한 일자 – 1 일로 설정한다. 또한, 사용가능포인트는 신용카드별로 현재까지 적립 또는 사용한 마일리지 포인트를 합산한다.

**SQL:**

```
CREATE TABLE 사용가능포인트테이블
AS
SELECT 카드번호 AS 카드번호,
       포인트적립사용일자 AS 유효시작일자,
       NVL(TO_CHAR(TO_DATE(LEAD(포인트적립사용일자)
            OVER (PARTITION BY 카드번호 ORDER BY 포인트적립사용일자),
          'YYYYMMDD') - 1, 'YYYYMMDD'), '99991231') AS 유효종료일자,
       SUM(SUM(적립사용포인트)) OVER (PARTITION BY 카드번호
                                      ORDER BY 포인트적립사용일자) AS 사용가능포인트
 FROM 마일리지테이블
GROUP BY 카드번호, 포인트적립사용일자;
```

결과 테이블 (사용가능포인트테이블)

| 카드번호 | 유효시작일자 | 유효종료일자 | 사용가능포인트 |
|---|---|---|---|
| A1000876 | 20011006 | 20011009 | 20 |
| A1000876 | 20011010 | 20011029 | 140 |
| A1000876 | 20011030 | 99991231 | 640 |
| A2000231 | 20011009 | 20011031 | 300 |
| A2000231 | 20011101 | 20011204 | 700 |
| A2000231 | 20011205 | 20020109 | 500 |
| A2000231 | 20020110 | 99991231 | 150 |
| A3000567 | 20011001 | 20011009 | 200 |
| A3000567 | 20011010 | 20011031 | 400 |
| A3000567 | 20011101 | 99991231 | 500 |

**② 비어있는 순번 찾기**

위에서 예로 들었던 마일리지테이블의 순번은 신용카드별 포인트적립사용일자별 순번이다. 즉, 특정 신용카드에 대해 동일 일자에 마일리지 포인트를 적립 또는 사용한 횟수가 다수인 경우 순번을 1 부터 차례로 증가시킨다. 신용카드의 마일리지 포인트에 대한 적립 또는 사용 이력에 대한 삭제가 가능해서 중간에 비어 있는 순번이 존재하고 이를 찾아서 조회해야 하는 요구를 만족하는 SQL 을 작성해야 한다고 가정해보자.

우선 대상 마일리지테이블의 데이터는 다음과 같다.

| 카드번호 | 포인트적립 사용일자 | 순번 | 적립사용 포인트 | |
|---|---|---|---|---|
| A1000876 | 20011006 | 1 | 100 | |
| A1000876 | 20011006 | 2 | -80 | |
| A1000876 | 20011010 | 1 | 120 | |
| A1000876 | 20011030 | 1 | 500 | |
| A2000231 | 20011009 | 1 | 300 | |
| A2000231 | 20011101 | 1 | 400 | |
| A2000231 | 20011205 | 1 | -500 | |
| A2000231 | 20011205 | 2 | 300 | |
| A2000231 | 20020110 | 1 | -350 | ▶2 ~ 4)까지 비어 있음 |
| A2000231 | 20020110 | 5 | -100 | |
| A2000231 | 20020406 | 10 | 100 | ▶1 ~ 9)까지 비어 있음 |
| A3000567 | 20011001 | 2 | 200 | ▶1이 비어 있음. |
| A3000567 | 20011010 | 1 | 600 | |
| A3000567 | 20011010 | 2 | -100 | |
| A3000567 | 20011010 | 3 | -300 | |
| A3000567 | 20011101 | 1 | 100 | |

위의 테이블에 대해 비어 있는 순번을 카드번호와 포인트적립사용일자별로 출력하는 SQL 은 다음과 같다.

```
select a.카드번호,
       a.포인트적립사용일자,
       b.no
  from (select 카드번호, 포인트적립사용일자, 순번,
               lag(순번, 1, 0) over (partition by 카드번호,
                     포인트적립사용일자 order by 순번) as 이전순번
          from 마일리지테이블) a,
               copy_t b              -- 비어 있는 순번 재생을 위한 복제용 테이블
 where a.순번 > a.이전순번 + 1
   and b.no between a.이전순번 + 1 and a.순번 - 1;
```

결과 집합 :

| 카드번호 | 포인트적립사용일자 | 순번 |
|---|---|---|
| A2000231 | 20020110 | 2 |
| A2000231 | 20020110 | 3 |
| A2000231 | 20020110 | 4 |
| A2000231 | 20020406 | 1 |
| A2000231 | 20020406 | 2 |
| A2000231 | 20020406 | 3 |
| A2000231 | 20020406 | 4 |
| A2000231 | 20020406 | 5 |
| A2000231 | 20020406 | 6 |
| A2000231 | 20020406 | 7 |
| A2000231 | 20020406 | 8 |
| A2000231 | 20020406 | 9 |
| A3000567 | 20011001 | 1 |

## 7. 게시판 Query 및 Directory & File 구조 Query

### 1) 게시판(응답 글 없는) Query

**PURPOSE**

Analytic function 등 신기술을 활용한 웹 게시판에서의 부분범위처리를 구현함으로써 data access 효율을 극대화시킬 수 있는 방안 제시

**SCOPE & APPLICATION**

인터넷이 대중화되고 양방향화 되면서 어떤 인터넷 사이트를 망라하고 게시판이 구현되지 않는 곳은 거의 없다. 그러나, connectionless(stateless)의 기본 성격을 가지는 웹의 구조적인 문제 때문에, 웹은 DB 의 이전 state 를 기억할 수 없는 근본적인 한계가 있다. 물론, 이를 해결하기 위하여 middleware(application server)를 도입한 3-Tier 환경으로의 전환이나 Java 등의 전혀 새로운 architecture 가 제시되었다. 그러나, 현실적으로 middleware 는 비용부담의 측면을 차치하고서라도 network balancing 의 역할 이상을 기대하기 어렵고 Java 는 속도 면에 많은 문제를 안고 있다. 또한, 더 근본적으로 웹의 게시판은 'random page jump' 속성을 가지고 있어 일반 클라이언트 프로그램에서 page up/down 시의 부분범위처리와는 성격이 전혀 다르다.
이러한 현실적인 어려움 때문에 대부분의 게시판은 사용자 액션(page click) 이 들어올 때마다, 게시판 DB table 전체를 읽고 나서 필요한 부분만 잘라서 보여주고 있다. 그러나, 이러한 방식은 게시판의 글 건수가 증가하면 증가할수록 응답속도가 떨어질 수밖에 없는 뻔한 결론을 가지고 있다.
본 토픽에서는, 현재 운영 중인 한 포탈사이트(운영환경: OS - Linux, Web - Apache PHP, DB - Oracle 8.1.6)에서 웹 게시판의 부분범위처리를 구현하여 실제로 성공적으로 적용되고 있는 방법을 공개하기로 한다. 본 토픽에서는 응답이 없는 게시판(공지사항 등...)을 대상으로 한 방법을 공개하고, 다음 토픽에서 응답이 있는 게시판에 대해서 다루기로 한다.

**KEY IDEA**

웹 게시판, 부분범위처리, page, screen, analytic function

**SUPPOSITION**

앞에서도 언급한 바와 같이, 웹 게시판은 DB 의 이전 state 를 기억할 수 없다는 점과 'random page jump'의 속성을 가지고 있다는 것이 부분범위처리를 가로막는 주요 원인이다.
그러나, 웹의 구조와 웹 게시판의 전형적인 인테페이스 구조를 자세히 살펴보면, 이러한 문제가 전혀 해결할 수 없는 문제가 아님을 알 수 있다.

- 웹이 DB 의 이전 state 를 기억할 수 없다는 점에 대해서

  웹이 기본적으로 connectionless 구조를 가지는 것은 사실이지만, 이를 보완할 수 있는 구조가 전혀 없는 것은 아니다. 그 대표적인 사례가 쿠키(Cookie) 이다. 쿠키는 사용자 로그온 시부터 로그아웃 시까지 사용자 별로 지속적으로 참조해야 하는 정보를 담을 수 있는 오브젝트이다. 또한, 제한적이기는 하지만, cgi parameter 도 이러한 역할을 할 수 있다. 이를 웹 게시판에 이용하면, 현 page view 시에 읽은 데이터의 시작점과 끝점 등의 필요한 정보를 다음 page view 시에 이용할 수 있다.

- 웹 게시판의 'random page jump' 속성에 대해서

  웹 게시판은 1 page 에서 2 page 로만 갈 수 있는 것이 아니고 바로 10 page 로 jump 하여 게시 글을 볼 수도 있는 구조를 가지고 있다. 그러나, 웹 게시판의 전형적인 인터페이스 구조를 곰곰이 살펴보면, page down/up 에 해당하는 특성을 찾을 수 있다. 대부분의 게시판은 다음과 같은 형태로 되어 있다.

  ◀ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 ▶

  한 번에 보여줄 수 있는 page 들의 단위를 screen 이라고 하고, 한 screen 은 10 page 로 구성되어 있다고 가정하자. 그러면, 우리는 "1 page 에서 10page 로는 바로 갈 수 있지만, 1 page 에서 31 page 로는 바로 갈 수 없다"는 중요한 점을 발견할 수 있다. 즉, page 는 random jump 를 하지만 screen 은 순차적인 page down/up 구조와 동일하다는 것이다.

  이와 같이, 웹 게시판의 부분범위처리를 가로막는 두 요인에 대한 해결책은 분명히 있다. 그러면, 어떻게 구현할 것인가?
  (앞으로, 한 screen 은 10page, 1 page 는 10 개의 글로 구성되어 있다는 전제로 서술하겠다.)

**① 기본적인 mechanism**

웹 게시판은 random jump 의 범위가 전체 page 가 아니라 해당 screen 내 라는 중요한 점을 발견하였다. random 은 말 그대로 어디로 갈 지 모르기 때문에, 해당 범위를 다 읽을 수밖에 없다. 그러나, 한 screen 에 해당하는 데이터 량만 읽으면 된다. 만약, 1 page 가 10 건의 글로 구성되어 있다는 100 건을 읽고 그 중 해당 page 에 해당하는 10 건을 뿌려주는 것이다. 이와 같이 하면, 100 건 중에 10 건밖에 써 먹지는 못하는 비효율은 어쩔 수 없다 하더라도, 게시판이 몇 개의 글을 가지던 간에 원하는 page 를 읽기 위해서 항상 100 건만 읽으면 된다는 결론을 얻을 수 있 다.

**② 부분범위 처리를 위해 무슨 정보가 필요하나?**

우선, 현 page 가 속한 screen 의 시작점과 끝점의 정보를 cookie 나 cgi parameter 에 담아야 한다. 웹 게시판은 게시된 순서의 역순으로 출력하는(응답 글이 있는 게시판은 좀 다르지만) 것이 일반 적이므로 screen 의 시작점(가장 큰 글 번호)는 screen up 시에 출발점으로 활용해야 하고 screen 의 끝점(가장 작은 글 번호)는 screen down 시에 출발점으로 활용해야 한다.
또한, 현 screen 에 해당하는 글 수를 알아야 한다.(이는 cookie 나 cgi parameter 에 담을 필요는 없다) 왜냐하면, 몇 번째 screen 이 마지막 screen 이 될 지 모르는 상태에서 마지막 screen 에서는 몇 page 를 뿌려야 할 지를 알아야 하기 때문이다. 또한, 마지막 screen 에서는 '▶'를 disable 시켜야 한다. 이번 screen 이 마지막 screen 인지 아닌지를 알려면, 100 건만 읽어서는 알 수 없고 그 100 건 다 음에 글이 있느냐? 없느냐? 를 알아야 하기 때문에 101 건을 읽어 야 한다. 그래서, stop key 를 이용하여 101 건으로 끊었을 때, 실 제로 101 건을 읽으면 다음 screen 이 존재하는 것이고 101 건보다 작은 건수를 읽으면 마지막 screen 이 되는 것이다. 마지막 screen 으로 판명되면 실제 읽은 건수로 몇 page 까지 뿌려야 할 지를 알 수 있다.

**③ Table Creation & Inserting Sample Data (crtb_NON_REPLY_BBS.sql)**

```
CREATE TABLE NON_REPLY_BBS (
        nrb_seq          NUMBER(10) NOT NULL,
        userid           VARCHAR2(8) NOT NULL,
        subject          VARCHAR2(64),
        regdate          DATE,
        view_cnt         NUMBER(10)
);

ALTER TABLE NON_REPLY_BBS ADD CONSTRAINT NON_REPLY_BBS_PK PRIMARY KEY ( nrb_seq );

COMMENT ON TABLE NON_REPLY_BBS IS '응답글 없는 게시판';
COMMENT ON COLUMN NON_REPLY_BBS.nrb_seq IS '게시글 번호';
COMMENT ON COLUMN NON_REPLY_BBS.userid IS 'User ID';
COMMENT ON COLUMN NON_REPLY_BBS.subject IS '글 제목';
COMMENT ON COLUMN NON_REPLY_BBS.regdate IS '등록일';
COMMENT ON COLUMN NON_REPLY_BBS.view_cnt IS '조회수';

insert into NON_REPLY_BBS
select rownum+1, 'USER_'||round(DBMS_RANDOM.value(1,100)), 'TITLE'||(rownum+1),
       sysdate-(1000-rownum), round(DBMS_RANDOM.value(1,10000))
  from dual
  connect by rownum < 1000;

commit;
```

**④ Selecting Data (select_NON_REPLY_BBS.sql)**

```
-- Initial Screen or Screen Down
select nrb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select /*+ index_desc(a non_reply_bbs_pk) */
                nrb_seq, userid, subject, regdate, view_cnt,
                min(nrb_seq) over ( ) as min_seq,
                max(nrb_seq) over ( ) as max_seq,
                count(*) over ( ) as cnt, rownum rnum
           from non_reply_bbs a
          where &SCREEN_DOWN = 1
            and nrb_seq <= decode(trunc(&PAGE_NO/11),0,9999999999,&MIN_SEQ)
            and rownum <= 101
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
union all
-- Random Page Jump in 1 Screen
select nrb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select /*+ index_desc(a non_reply_bbs_pk) */
                nrb_seq, userid, subject, regdate, view_cnt,
                min(nrb_seq) over ( ) as min_seq,
                max(nrb_seq) over ( ) as max_seq,
                count(*) over ( ) as cnt, rownum rnum
           from non_reply_bbs a
          where &SCREEN_DOWN = 0
            and nrb_seq <= decode(trunc(&PAGE_NO/11),0,9999999999,&MAX_SEQ)
            and rownum <= 101
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
union all
-- Screen Up
select nrb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select nrb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt, rownum rnum
           from ( select /*+ index (a non_reply_bbs_pk) */
                         nrb_seq, userid, subject, regdate, view_cnt,
                         min(nrb_seq) over (order by nrb_seq desc rows between unbounded
preceding and unbounded following) as min_seq,
                         max(nrb_seq) over ( ) as max_seq,
                         count(*) over ( ) as cnt
                    from non_reply_bbs a
                   where &SCREEN_DOWN = -1
                     and nrb_seq >= &MAX_SEQ
                     and rownum <= 101
                )
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
```

본 SQL 은 nrb_seq(글 번호)의 순서와 글 게시시각의 순서가 일치한다는 전제 하에서 정상적으로 동작한다. 응답 글이 없는 게시판은 일반적으로 이러한 성질을 가진다. 즉, 게시시각의 순서로 글 번호가 매겨지며, 이의 역순으로 출력된다는 것이다.

본 SQL 에서 cookie 나 cgi parameter 에 담아야 할 정보는 min_seq 와 max_seq 이다. 이 두 정보만 있으면, 다음 screen 의 시작점을 쉽게 알 수 있다.

본 SQL 에서 특히 주의해야 할 점은, 첫 번째 screen(1 page - 10 page)를 뿌릴 때는 항상 &SCREEN_DOWN = 1 (page down mode)로 주고 뿌려야 한다는 것이다. 왜냐하면, 웹 게시판은 우리가 글 을 보는 사이에도 끊임없이 새로운 글이 올라온다는 사실 때문 이다. 이렇게 하지 않으면, 글을 보는 사이에 올라온 새로운 글은 게시판을 처음부터 다시 들어오지 않는 한 절대 볼 수 없다 는 문제가 생긴다. 물론, 대가는 있다. 두 번째 screen 에서 첫 번째 screen 으로 갈 때는 새로 게시된 글 수만큼 기존 글 을 볼 수 없다는 것이다. 그러나, 이 정도는 감수할 수 있다. 보이지 않는 글은 이미 오래된 글이며, 다시 두 번째 screen 으로 이동할 때, 보이기 때문이다.

단, 두 번째 screen 에서 첫 번째 screen 으로 이동 시 &SCREEN_DOWN = -1 로 뿌리는 방법도 선택할 수 있다. 이럴 경우 게시판을 보고 있는 동안 새로 올라온 글 수만큼은 인지할 수 없다.
그렇지만, 첫 번째 screen 에서 random page jump 를 한 번만 수행하게 되면 &SCREEN_DOWN = 0 이 설정되어 다음의 조건으로 인해 새로운 글이 모두 포함된 상태가 된다.

nrb_seq <= decode(trunc(&PAGE_NO/11),0,9999999999,&MAX_SEQ)

본 SQL 은 screen 의 시작점과 끝점, screen 의 글 건수를 구하기 위해 analytic function 을 사용하였다. 이는 오라클 8.1.6 이상의 version 에서만 동작하므로, version 이 낮은 오라클 사용자나 다른 DBMS 사용자는 사용이 불가하다. 그러나, 걱정할 필요는 없다. 테이블을 101 건씩 두 번 읽으면 된다. 한 번은 일반적인 글 정보를 출력하기 위해, 다른 한 번은 screen 의 시작점과 끝점, screen 의 글 건수를 구하기 위해 사용하면 된다. 이렇게 하더라도, 매번 게시판 전체를 읽는 비효율 보다는 훨씬 낫다.

**<demo>**

```
SQL> COLUMN nrb_seq FORMAT 999999
SQL> COLUMN subject FORMAT A20
SQL> COLUMN min_seq FORMAT 9999999999
SQL> COLUMN max_seq FORMAT 9999999999

SQL> SET VERIFY OFF

SQL> ACCEPT SCREEN_DOWN PROMPT 'SCREEN_DOWN: '
SQL> ACCEPT PAGE_NO PROMPT 'PAGE_NO: '
SQL> ACCEPT MIN_SEQ PROMPT 'MIN_SEQ: '
SQL> ACCEPT MAX_SEQ PROMPT 'MAX_SEQ: '

SQL> @select_NON_REPLY_BBS
SCREEN_DOWN: 1
PAGE_NO: 1
MIN_SEQ: 0
MAX_SEQ: 0
```

**최초 1 page display**

| NRB_SEQ | USERID | SUBJECT | REGDATE | VIEW_CNT | MIN_SEQ | MAX_SEQ | CNT |
|---|---|---|---|---|---|---|---|
| 1000 | USER_45 | TITLE1000 | 12-DEC-12 | 6905 | 900 | 1000 | 101 |
| 999 | USER_84 | TITLE999 | 11-DEC-12 | 5036 | 900 | 1000 | 101 |
| 998 | USER_28 | TITLE998 | 10-DEC-12 | 2314 | 900 | 1000 | 101 |
| 997 | USER_9 | TITLE997 | 09-DEC-12 | 832 | 900 | 1000 | 101 |
| 996 | USER_48 | TITLE996 | 08-DEC-12 | 7848 | 900 | 1000 | 101 |
| 995 | USER_56 | TITLE995 | 07-DEC-12 | 7111 | 900 | 1000 | 101 |
| 994 | USER_41 | TITLE994 | 06-DEC-12 | 8020 | 900 | 1000 | 101 |
| 993 | USER_9 | TITLE993 | 05-DEC-12 | 6972 | 900 | 1000 | 101 |
| 992 | USER_22 | TITLE992 | 04-DEC-12 | 4069 | 900 | 1000 | 101 |
| 991 | USER_71 | TITLE991 | 03-DEC-12 | 9556 | 900 | 1000 | 101 |

10 rows selected.

```
SQL> @select_NON_REPLY_BBS
SCREEN_DOWN: 0
PAGE_NO: 2
MIN_SEQ: 900
MAX_SEQ: 1000
```

**첫 번째 Screen에서 2page로 이동**

| NRB_SEQ | USERID | SUBJECT | REGDATE | VIEW_CNT | MIN_SEQ | MAX_SEQ | CNT |
|---|---|---|---|---|---|---|---|
| 990 | USER_78 | TITLE990 | 02-DEC-12 | 8233 | 900 | 1000 | 101 |
| 989 | USER_37 | TITLE989 | 01-DEC-12 | 6882 | 900 | 1000 | 101 |
| 988 | USER_61 | TITLE988 | 30-NOV-12 | 3243 | 900 | 1000 | 101 |
| 987 | USER_96 | TITLE987 | 29-NOV-12 | 8396 | 900 | 1000 | 101 |
| 986 | USER_28 | TITLE986 | 28-NOV-12 | 693 | 900 | 1000 | 101 |
| 985 | USER_49 | TITLE985 | 27-NOV-12 | 9379 | 900 | 1000 | 101 |
| 984 | USER_71 | TITLE984 | 26-NOV-12 | 8748 | 900 | 1000 | 101 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 983 | USER_34 | TITLE983 | 25-NOV-12 | 1503 | 900 | 1000 | 101 |
| 982 | USER_96 | TITLE982 | 24-NOV-12 | 6765 | 900 | 1000 | 101 |
| 981 | USER_57 | TITLE981 | 23-NOV-12 | 949 | 900 | 1000 | 101 |

10 rows selected.

```
SQL> @select_NON_REPLY_BBS
SCREEN_DOWN: 1
PAGE_NO: 11
MIN_SEQ: 900
MAX_SEQ: 1000
```

**첫 번째 Screen에서 두 번째 Screen으로 Screen Down**

| NRB_SEQ | USERID | SUBJECT | REGDATE | VIEW_CNT | MIN_SEQ | MAX_SEQ | CNT |
|---|---|---|---|---|---|---|---|
| 900 | USER_76 | TITLE900 | 03-SEP-12 | 1711 | 800 | 900 | 101 |
| 899 | USER_76 | TITLE899 | 02-SEP-12 | 4841 | 800 | 900 | 101 |
| 898 | USER_82 | TITLE898 | 01-SEP-12 | 5748 | 800 | 900 | 101 |
| 897 | USER_72 | TITLE897 | 31-AUG-12 | 8826 | 800 | 900 | 101 |
| 896 | USER_88 | TITLE896 | 30-AUG-12 | 9045 | 800 | 900 | 101 |
| 895 | USER_66 | TITLE895 | 29-AUG-12 | 4256 | 800 | 900 | 101 |
| 894 | USER_17 | TITLE894 | 28-AUG-12 | 9850 | 800 | 900 | 101 |
| 893 | USER_42 | TITLE893 | 27-AUG-12 | 362 | 800 | 900 | 101 |
| 892 | USER_57 | TITLE892 | 26-AUG-12 | 6268 | 800 | 900 | 101 |
| 891 | USER_41 | TITLE891 | 25-AUG-12 | 7132 | 800 | 900 | 101 |

10 rows selected.

```
SQL> @select_NON_REPLY_BBS
SCREEN_DOWN: -1
PAGE_NO: 10
MIN_SEQ: 800
MAX_SEQ: 900
```

**두 번째 Screen에서 첫 번째 Screen으로 Screen Up**

| NRB_SEQ | USERID | SUBJECT | REGDATE | VIEW_CNT | MIN_SEQ | MAX_SEQ | CNT |
|---|---|---|---|---|---|---|---|
| 910 | USER_51 | TITLE910 | 13-SEP-12 | 8349 | 900 | 1000 | 101 |
| 909 | USER_26 | TITLE909 | 12-SEP-12 | 9578 | 900 | 1000 | 101 |
| 908 | USER_96 | TITLE908 | 11-SEP-12 | 7355 | 900 | 1000 | 101 |
| 907 | USER_90 | TITLE907 | 10-SEP-12 | 6802 | 900 | 1000 | 101 |
| 906 | USER_93 | TITLE906 | 09-SEP-12 | 4352 | 900 | 1000 | 101 |
| 905 | USER_34 | TITLE905 | 08-SEP-12 | 4397 | 900 | 1000 | 101 |
| 904 | USER_3 | TITLE904 | 07-SEP-12 | 2491 | 900 | 1000 | 101 |
| 903 | USER_91 | TITLE903 | 06-SEP-12 | 2268 | 900 | 1000 | 101 |
| 902 | USER_23 | TITLE902 | 05-SEP-12 | 6069 | 900 | 1000 | 101 |
| 901 | USER_5 | TITLE901 | 04-SEP-12 | 1647 | 900 | 1000 | 101 |

10 rows selected.

### 2) 게시판(응답 글 있는) Query

**PURPOSE**
Analytic function 등 신기술을 활용한 웹 게시판에서의 부분범위처리를 구현함으로써 data access 효율을 극대화시킬 수 있는 방안 제시

**SCOPE & APPLICATION**
웹 게시판과 DB 와의 연동에 관한 전반적인 문제점은 앞의 '응답 글이 없는 웹 게시판의 부분범위처리'에서 이미 언급하였다. 본 토픽에서는 응답 글이 있는 웹 게시판에서의 부분범위처리에 대해서 살펴보겠다. 앞 토픽에서 언급한 바와 같이, 응답 글이 있는 게시판은 글이 게시되는 순서와 출력되는 순서가 일치하지 않기 때문에 응답 글이 없는 게시판에 비해 생각해야 할 것이 훨씬 많다. 그러나, 이것 또한 약간의 additional 정보와 순환구조의 이해가 바탕이 되면, 1-SQL 로 처리가 가능하다.

**KEY IDEA**
웹 게시판, 부분범위처리, page, screen, analytic function

**SUPPOSITION**
앞에서도 언급한 바와 같이, 응답 글이 있는 웹 게시판은 글의 게시순서와 출력순서가 다를 수밖에 없다. 응답 글이 없는 게시판은 글이 게시되는 순서대로 글 번호(PK)가 매겨지고 그것의 역순(최신 것부터)으로 출력하면 그만이다. 그러나, 응답 글이 있는 게시판의 경우, 원글에 대한 답이 한참 후에 달리더라도 원글 바로 밑에 뿌려져야 하기 때문에 단순히 글 번호 순으로 뿌리게 되면 게시판은 뒤죽박죽이 될 수밖에 없다. 또한, 응답 글에 대한 응답 글이 또 달릴 수 있으므로 그 depth 가 어디까지 갈지 아무도 모른다. 그러므로, 이 문제는 procedural 하게 풀든, SQL 로 풀든 간에 순환(recursive)구조로 풀어야 한다.

- 글의 게시 순서와 출력순서가 다른 점에 대해서
  글의 게시순서에 대한 정보는 응답 글이 있는 게시판도 글 번호가 가지고 있다. 그러나, 글의 출력순서는 일반적인 게시판 테이블 어디에도 없다. 일반적으로, 응답 글이 있는 게시판 테이블은 응답 글의 원글은 무엇인가? 하는 정보는 담고 있다. 또한, 출력할 때, indentation 을 위해서 글의 depth(level) 또한 가지고 있다. 보통은 이 두 가지 정보를 가지고 프로그램에서 복잡하게 처리하여 출력순서에 맞는 구조를 만들어 낸다. 그러나, 이것은 상당히 구현하기 힘든 어려운 로직임에 틀림없다. 어떻게 이 문제를 해결할 것인가? 해결방법은 논리적으로는 간단하다. 즉, 글의 출력순서에 관한 정보를 테이블이 가지고 있으면 된다. 응답 글은 시조 원글 (가장 상위 level 의 원글) 단위로 묶여져 뿌려지게 된다. 그러므로, 시조 원글에 달린 글들 사이에 출력순서정보를 저장하면 되는 것이다. 예를 들어, 어떤 글에 대한 시조 원글의 글 번호를 v_group, 시조 원글 내의 출력순서정보를 v_order 라고 하면,

  ```
  select * from t_bbs order by v_group desc, v_order
  ```

  의 순서로 뿌리면 바로 출력순서인 것이다. 그러므로, '특정 글에 대한 시조 원글의 글 번호'와 '시조 원글 내의 출력순서정보'라는 두 가지 additional 정보만 첨가한다면, 출력제어가 훨씬 간단해 진다.
- 글의 depth(level)에 대해서
  앞에서 언급한 출력순서정보를 따로 가지고 있으면 출력 시에는 순환구조를 활용할 필요가 없으나, 글 삭제 시에는 여전히 필요하다. '글을 삭제하면 그 밑에 달린 글들도 같이 삭제한다'가 일반 적이므로 몇 대에 걸쳐 있을지도 모를 자손들을 다 없애기 위해서 순환구조의 활용은 필수적이다. 오라클은 SQL 에서 순환구조(connect by...start with)를 지원한다. 그러므로, 오라클을 사용할 경우, 순환구조의 처리가 SQL 내에서 가능하다.

이제, 응답 글이 있는 웹 게시판의 부분범위처리와 글입력/글삭제 시의 maintenance 방법, 그리고, 기존 게시판 테이블에 출력순서정보 (v_group, v_order)를 담는 방법에 대해 언급하겠다.

게시판 REPLY_BBS 의 테이블 구조는 다음과 같다.

```
CREATE TABLE REPLY_BBS (
    rb_seq        NUMBER(10) NOT NULL,
    userid        VARCHAR2(8) NOT NULL,
    subject       VARCHAR2(64),
    regdate       DATE,
    view_cnt      NUMBER(10),
    p_rb_seq      NUMBER(10),
    v_depth       NUMBER(10),
    v_group       NUMBER(10),
    v_order       NUMBER(6)
);

COMMENT ON TABLE REPLY_BBS IS '응답글 있는 게시판';
COMMENT ON COLUMN REPLY_BBS.rb_seq IS '게시글 번호';
COMMENT ON COLUMN REPLY_BBS.userid IS 'User ID';
COMMENT ON COLUMN REPLY_BBS.subject IS '글 제목';
COMMENT ON COLUMN REPLY_BBS.regdate IS '등록일';
COMMENT ON COLUMN REPLY_BBS.view_cnt IS '조회수';
COMMENT ON COLUMN REPLY_BBS.p_rb_seq IS '상위글 번호';
COMMENT ON COLUMN REPLY_BBS.v_depth IS '글의 Depth';
COMMENT ON COLUMN REPLY_BBS.v_group IS '시조원글 번호';
COMMENT ON COLUMN REPLY_BBS.v_order IS '그룹내 출력 순서';
```

**① 부분범위 처리**

글의 출력은 응답 글이 없는 게시판의 부분범위처리와 기본적인 mechanism 이 일치한다. 그러나, 중요한 차이점이 있는 응답 글이 없는 경우는 screen/page 의 처리기준(출력순서의 기준)이 글 번호 (nrb_seq)였지만, 이 경우는 v_group+v_order 가 되어야 한다. 이렇게 두 컬럼 이상이 출력의 기준이 될 때는 v_group||v_order 와 같은 형식으로 크기 비교를 해야 한다. 왜냐하면, 각각의 크기가 크다고 전체가 큰 것은 절대로 아니기 때문이다. 그러므로, 여기서는 오라클 8.1.6 에 있는 FBI(Function Based Index)를 활용하여 다음과 같은 인덱스를 만든다.

```
CREATE UNIQUE INDEX REPLY_BBS_FBI1 ON REPLY_BBS
                                    (LPAD(TO_CHAR(v_group),10,'0')||TO_CHAR(999999-v_order));
```

더불어, Primary Key 와 Index 를 생성한다.

```
ALTER TABLE REPLY_BBS ADD CONSTRAINT REPLY_BBS_PK PRIMARY KEY ( rb_seq );
CREATE UNIQUE INDEX REPLY_BBS_IDX1 ON REPLY_BBS (p_rb_seq, rb_seq);
```

**② Selecting Data (select_REPLY_BBS.sql)**

```
select lpad(' ',2*v_depth)||rb_seq rb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select /*+ index_desc (a reply_bbs_fbi1) */
                rb_seq, userid, subject, regdate, view_cnt, v_depth,
                min(lpad(to_char(v_group),10,'0')||to_char(999999-v_order)) over ( ) as min_seq,
                max(lpad(to_char(v_group),10,'0')||to_char(999999-v_order)) over ( ) as max_seq,
                count(*) over ( ) as cnt, rownum rnum
           from reply_bbs a
          where &SCREEN_DOWN = 1
            and lpad(to_char(v_group),10,'0')||to_char(999999-v_order) <=
decode(trunc(&PAGE_NO/11),0,'9999999999999999',&MIN_SEQ)
            and rownum <= 101
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
union all
select lpad(' ',2*v_depth)||rb_seq rb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select /*+ index_desc (a reply_bbs_fbi1) */
                rb_seq, userid, subject, regdate, view_cnt, v_depth,
                min(lpad(to_char(v_group),10,'0')||to_char(999999-v_order)) over ( ) as min_seq,
                max(lpad(to_char(v_group),10,'0')||to_char(999999-v_order)) over ( ) as max_seq,
                count(*) over ( ) as cnt, rownum rnum
           from reply_bbs a
          where &SCREEN_DOWN = 0
            and lpad(to_char(v_group),10,'0')||to_char(999999-v_order) <=
decode(trunc(&PAGE_NO/11),0,'9999999999999999',&MAX_SEQ)
            and rownum <= 101
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
union all
select lpad(' ',2*v_depth)||rb_seq rb_seq, userid, subject, regdate, view_cnt, min_seq, max_seq, cnt
  from ( select rb_seq, userid, subject, regdate, view_cnt, v_depth, min_seq, max_seq, cnt, rownum
rnum
           from ( select /*+ index (a reply_bbs_fbi1) */
                         rb_seq, userid, subject, regdate, view_cnt, v_depth,
                         min(lpad(to_char(v_group),10,'0')||to_char(999999-v_order))
                           over (order by lpad(to_char(v_group),10,'0')||to_char(999999-v_order) desc
                           rows between unbounded preceding and unbounded following) as min_seq
                         max(lpad(to_char(v_group),10,'0')||to_char(999999-v_order)) over ( ) as max_seq,
                         count(*) over ( ) as cnt
                  from reply_bbs a
                 where &SCREEN_DOWN = -1
                   and lpad(to_char(v_group),10,'0')||to_char(999999-v_order) >= &MAX_SEQ
                   and rownum <= 101
                 )
       )
 where rnum between 10*(decode(mod(&PAGE_NO,10),0,9,mod(&PAGE_NO,10)-1))+1
                and 10*decode(mod(&PAGE_NO,10),0,10,mod(&PAGE_NO,10))
```

FBI 를 사용할 수 없는 상황의 개발자라면, 다른 방법을 강구해야 한다. 물론, 방법은 있다. 약간의 비효율(최대 2 배(202 건)를 읽어야 한다.)이 있고, SQL 이 길어지는 단점이 있긴 하지만, 여전히 전체를 읽는 비효율에 비해서는 탁월한 방법이다. 이것은 'solution warehouse -> application development -> 부분범위처리'에 올라온 글들을 참조하기 바란다.

### ③ Inserting & Deleting

글 입력 시는 게시판의 출력정보를 저장하는 v_group, v_order 정보도 같이 만들어 주어야 한다. 그러나, 이것은 간단한 작업이다. v_group 은 시조 원글의 글 번호이므로 원글의 v_group 과 같은 값으로 채워 넣으면 되고, v_order 는 '답글은 원글 바로 밑에 달린다'는 원칙이 있으면(보통 게시판은 이것이 일반적인 원칙이다.) 원글의 v_order+1 이 답글의 v_order 값이 된다. 물론, 그 후의 글들의 v_order 값도 1 씩 증가시켜야 한다.

```
update REPLY_BBS set v_order = v_order+1
  where v_group = :var_v_group                -- :var_v_group : 원글의 v_group
    and v_order > :var_v_order                -- :var_v_order : 원글의 v_order

insert into REPLY_BBS (v_group, v_order) values (:var_v_group, :var_v_order+1)
```

주의해야 할 것은, insert 전에 update 가 되어야 한다는 것이다. 그렇지 않으면, v_order 값이 중복되기 때문이다.

글 삭제 시는 순환구조로 대상글의 하위 답글까지 다 지워야 한다.

```
select count(*) into :var_cnt
  from REPLY_BBS
connect by prior rb_seq = p_rb_seq
start with rb_seq = :var_rb_seq          -- 지워질 글의 수 구하기 (:var_rb_seq : 삭제될 글의 rb_seq)

delete from REPLY_BBS
where rb_seq in ( select rb_seq
                    from REPLY_BBS
                  connect by prior rb_seq = p_rb_seq
                  start with rb_seq = :var_rb_seq )        -- 글 지우기

update REPLY_BBS set v_order = v_order - :var_cnt
where v_group = :var_v_group
  and v_order > :var_v_order                    -- v_order 값 조정 (:var_v_order: 삭제된 글의 v_order)
```

글 지우기 SQL 에 있는 connect by... start with 가 순환구조이다. 여기에는 반드시, p_rb_seq 로 시작하는 index 가 있어야 한다는 점을 명심하기 바란다. 물론, v_order 값이 지워진 글 수만큼 조정되어야 한다.

**④ 기존 게시판에서 출력순서정보(v_group, v_order)값을 얻는 방법**

기존 쓰고 있는 게시판에서, 위와 같이 부분범위처리를 하려고 해도 출력순서정보를 만들어내지 않으면 이 방법을 사용할 수 없다. 그러나, 이것 또한 1-SQL 로 만들어 낼 수 있다.

```
select v_group, rank() over (partition by v_group order by rnum) v_order
 from ( select /*+ index_desc (a reply_bbs_idx2) */
              ( select nvl(min(rb_seq), a.rb_seq)
                  from REPLY_BBS b
                 connect by prior p_rb_seq = rb_seq
                 start with b.rb_seq = a.p_rb_seq
               ) v_group, rownum rnum
          from REPLY_BBS a
        connect by prior rb_seq = p_rb_seq
        start with p_rb_seq = 0 ) ;

-- reply_bbs_idx2 : p_rb_seq
```

이 SQL 또 한 analytic function 과 select-list subquery 를 사용했기 때문에, 오라클 8.1.6 이상에서만 돌아간다. 순환구조를 두 번 사용하여 원하는 c_grp,c_step 값을 한 번에 찾아낼 수 있다. 어떤 원리인지에 대한 설명은 순환구조와 index descending 의 원리를 알면 파악할 수 있기 때문에 지면 관계상 생략하기로 한다.

### 3) Directory & File 구조 Query

```
SQL> desc dir_t
 Name                              Null?    Type
 --------------------------------- -------- -------------------------
 DIRID                             NOT NULL NUMBER
 DIRNAME                                    VARCHAR2(20)

SQL> desc file_t
 Name                              Null?    Type
 --------------------------------- -------- -------------------------
 DIRID                             NOT NULL NUMBER
 FILEID                            NOT NULL NUMBER
 FILENAME                                   VARCHAR2(40)

SQL> select * from dir_t;

   DIRID  DIRNAME
--------- --------------------
        1  1st Directory
        2  2nd Directory
        3  3rd Directory
        4  4th Directory
        5  5th Directory

SQL> select * from file_t;

   DIRID   FILEID  FILENAME
--------- -------- ----------------------------------------
        1        1 1st Directory, 1st File
        1        3 1st Directory, 2nd File
        1        4 1st Directory, 3rd File
        2        2 2nd Directory, 1st File
        2        3 2nd Directory, 2nd File
        4        2 4th Directory, 1st File
        4        3 4th Directory, 2nd File
        4        5 4th Directory, 3rd File
        4        8 4th Directory, 4th File
        4        9 4th Directory, 5th File
        5        4 5th Directory, 1st File
```

문제) 디렉토리(DIR_T)와 파일(FILE_T)관련 정보가 위와 같은 구조로 되어 있을 경우
다음과 같은 결과를 출력하기 위한 query 를 작성하시오.

```
    DIRID  TYPE       NAME
--------- -------- ------------------------------------------
        1 Dir      1st Directory
        1   |-File     1st Directory, 1st File
        1   |-File     1st Directory, 2nd File
        1   |-File     1st Directory, 3rd File
        2 Dir      2nd Directory
        2   |-File     2nd Directory, 1st File
        2   |-File     2nd Directory, 2nd File
        3 Dir      3rd Directory
        4 Dir      4th Directory
        4   |-File     4th Directory, 1st File
        4   |-File     4th Directory, 2nd File
        4   |-File     4th Directory, 3rd File
        4   |-File     4th Directory, 4th File
        4   |-File     4th Directory, 5th File
        5 Dir      5th Directory
        5   |-File     5th Directory, 1st File
```

```
select a.dirid,
       decode(a.rnum, 0, 'Dir', b.num, 'Dir', '   |-File') type,
       decode(a.rnum, 0, a.dirname, b.num, a.dirname, '      '||a.filename) name
  from ( select x.dirid, y.fileid, x.dirname, y.filename,
                decode(y.fileid, null, 0, (min(y.fileid) over(partition by y.dirid)),1,y.fileid) rnum
           from dir_t x, file_t y
          where x.dirid = y.dirid(+) ) a,
       ( select num from cartesian where num <= 2 ) b
 where b.num <= decode(a.rnum,1,2,1)
 order by a.dirid, a.fileid;
```

```
select dirid,
       decode(fileid,NULL,'Dir',decode(no,2,'Dir','   |-File')) type,
       decode(fileid,NULL,dirname,decode(no,2,dirname,'   '||filename)) name
  from ( select t.no, s.dirid, s.fileid, s.dirname, s.filename, s.min_fileid
           from ( select x.dirid, y.fileid, x.dirname, y.filename,
                         min(y.fileid) over(partition by y.dirid order by x.dirid, y.fileid) min_fileid
                    from csj_dir_t x, csj_file_t y
                   where x.dirid = y.dirid(+)
                ) s, copy_t t
          where t.no <= decode(s.fileid,NULL,1,s.min_fileid,2,1)
       )
 order by dirid, no desc, fileid;
```