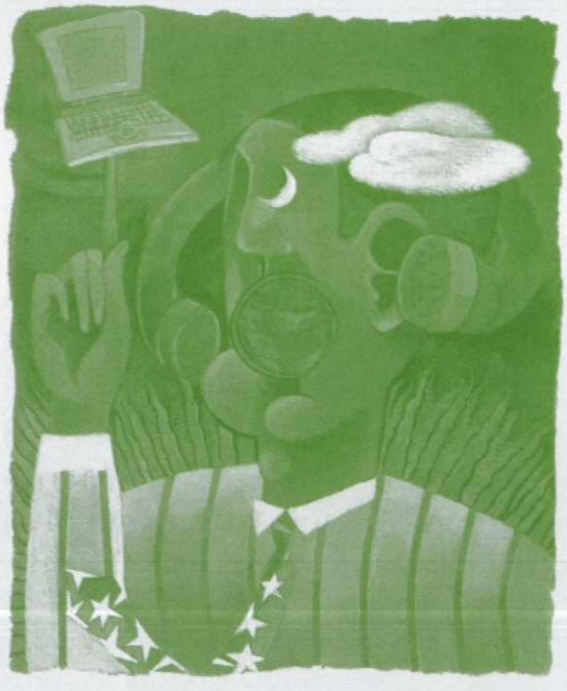
SQL 최적화가 성능 향상 출발점

SQL문은 하나 또는 여러 개의 테이블(집합)에서 사용자가 원하는 데이터(집합)를 찾아서 원하는 형태(집합)로 추출 및 가공 처리하는 것이다. 특히 SQL문에 조인, 뷰, 인라인뷰, 서브쿼리와 다양한 문자, 숫자, 그룹함수 등을 잘 활용하면 데이터를 쉽게 원하는 형태로 가공처리 할 수 있다. 따라서 SQL의 활용범위를 무궁무진하게 넓혀 나갈 수 있다고 생각한다. 따라서 이 글에서는 SQL문의 활용범위를 확장시킬 수 있는 원리와 활용 방법을 몇 가지 사례를 통해 설명하겠다.

■ 김동훈/ 삼성SDS 팀장

연재순서



관계형 데이터베이스에서 제공하는 함수를 최대한 활용

SQL문으로 데이터를 처리하는데 있어 가장 부족한 것이 'CASE BY CASE' 형태의 데이터를 처리하는 것이다. 이런 경우 가장 쉬운 방법은 절차형 언어를 사용하여 처리하는 것이다.

그러나 앞에서 설명한 것처럼 옵티마이져가 최적의 데이터 처리경로를 수립하는 단위는 SQL이므로 절차형 언어를 사용하여 처리하는 것보다 하나의 SQL문으로 처리하는 것이 유리하다. 이러한 'CASE BY CASE' 형태의 데이터 처리를 쉽게 SQL문장 하나로 처리할 수 있게 하는 요소들이 바로 관계형 데이터베이스에 있는 함수들이다

■ 그룹함수 MIN, MAX의 활용

가장 최근의 일자를 가진 로우를 추출하는 경우 또는 어떤 컬럼의 값이 가장 큰 로우를 추출하는 것은 아주 단순한 것 같지만 어떤 경우는 매우 까다롭고 수행속도에 많은 영향을 미친다.

다음의 예와 같은 경우는 간단한 아이디어를 추가함으로써 2개의 SQL문으로 처리하는 것을 하나의 SQL문으로 처리한 사례이다. 입금구분이 '1'인 것 중에서 가장 최근에 입금한 일자를 최종입금일로 구하고 '1'이 없으면 입금구분이 가장 큰 값 중에서 가장 최근에 입금한 일자를 최종입금일로 구하는 경우이다.

```
SELECT MAX(입금일) INTO :최종입금일 FROM 입금실적
WHERE 수주번호 = :suju_no AND 입금구분 ='1';
EXCEPTION
WHEN NO_DATA_FOUND THEN
SELECT MAX(입금일) INTO :최종입금일 FROM 입금실적
WHERE (수주번호,입금구분) = (SELECT :suju_no, MAX(입금구분)
                            FROM 입금실적
                            WHERE 수주번호 = :suju_no);
```

이것을 입금구분이 '1' 이면 가장 큰 값으로 변경하여 입금일과 결합한 후 MAX값을 취하면 자동적으로 가장 큰 입금구분의 입금일이 최종입금일이 되기 때문에 아래와 같은 SQL문으로 구현할 수 있다.

```
SELECT SUBSTR(MAX(DECODE(입금구분,'1','5')||입금일),2,8) INTO :최종
입금일  FROM  입금실적  WHERE 수주번호 = :suju_no;
```

또한 MAX, MIN 함수는 유일하게 모든 데이터 타입에 사용할 수 있는 그룹함수이기 때문에 1:M의 조인 후 '1'에 해당하는 컬럼의 값을 GROUP BY 절에 지정할 수 없는 경우의 문제점을 해결할 수 있다.

앞에서 보는 것과 같이 1:M의 관계를 가지는 집합을 조인하면 M개의 로우가 생긴다. 만일 1쪽에 있는 컬럼을 SUM하는 경우 잘못된 결과를 얻게 될 수 있다. 따라서 1쪽에 있는 컬럼은 MIN, MAX와 같은 함수를 사용하거나 GROUP BY 절에 기술해야만 한다.

그러나 GROUP BY절에 이러한 모든 컬럼들을 기술하게 되면 이 컬럼 중 NULL 값을 가지는 경우 NULL인 로우와 NULL이 아닌 로우가 각각의 GROUP으로 집계된다.

이때 모든 데이터 타입에 사용할 수 있는 MAX, MIN 함수를 사용하면 자동적으로 NULL 값은 처리대상에서 제외된다.

```
SELECT TAB1.성명, MIN(구분), MIN(단가), SUM(금액) FROM TAB1, TAB2
WHERE TAB1.성명= TAB2.성명
GROUP BY TAB1.성명
```

■ 여러가지 함수와 'DECODE' 함수를 함께 활용한 사례

'DECODE' 함수는 절차형 언어에서 사용하는 IF ...THEN ... ELSEIF ...END IF와 같은 기능을 가지고 있지만 절차형 언어에서 사용하는 '>', 'OR'와 같은 연산자를 사용할 수 없고 단지 '=' 형태로만 사용되는 단점이 있다. 그러나 다른 함수들을 잘 이용하면 쉽게 이러한 단점들을 해결할 수 있다. 다음과 같은 경우를 살펴보자.

정상적인 퇴근 시간은 18시이고 퇴근 시간이 21시 이전이면 시간당 1,000원이고 21시에서 24시까지는 시간당 3,000원의 수당을 지급하는 경우를 절차형 언어로 구현하면 아래와 같이 구현할 수 있다.

```
SELECT END_TIME INTO :e_time FROM DAILY
WHERE SABUN=:sabun AND JDATE = :jdate;
IF :e_time <='2100' THEN
SELECT CEIL( (TO_NUMBER(SUBSTR(END_TIME,1,2))*60 +
             TO_NUMBER(SUBSTR(END_TIME,3,2)) -
             TO_NUMBER(18)*60)/60 ) *1000  FROM DAILY
WHERE SABUN=:sabun AND JDATE = :jdate;
ELSIF :e_time > '2100'
SELECT 3*1000 + CEIL ((TO_NUMBER(SUBSTR(END_TIME,1,2))*60 +
                       TO_NUMBER(SUBSTR(END_TIME,3,2)) -
                       TO_NUMBER(21)*60)/60 ) *3000 FROM DAILY
WHERE SABUN=:sabun AND JDATE = :jdate;
END IF;
```

이것을 'DECODE'와 여러 개의 다른 함수와 함께 사용하면 다음과 같이 하나의 SQL문으로 최적화할 수 있다.

```
SELECT DECODE(SIGN(TO_NUMBER(SUBSTR(END_TIME,1,2))*60 +
       TO_NUMBER(SUBSTR(END_TIME,3,2)) - TO_NUMBER(21)*60),
         -1, CEIL( (TO_NUMBER(SUBSTR(END_TIME,1,2))*60 +
           TO_NUMBER(SUBSTR(END_TIME,3,2)) -
           TO_NUMBER(18)*60)/60 ) 1000,
```

```
                0, 3*1000 ,
1, 3*1000 + CEIL ((TO_NUMBER(SUBSTR(END_TIME,1,2))*60 +
                TO_NUMBER(SUBSTR(END_TIME,3,2)) -
                TO_NUMBER(21)*60)/60 ) *3000)
FROM DAILY
WHERE SABUN=:sabun AND JDATE = :jdate and END_TIME > '1800' ;
```

■ GROUP 함수들과 함께 'DECODE' 함수를 활용한 사례

앞에서 설명한 DECODE 함수의 활용 방법에 SUM, MAX, MIN, COUNT 등의 GROUP함수들을 함께 활용하게 되면 처리할 집합의 영역을 쉽게 정할 수 있고, 처리 종류나 추출하고 싶은 단위로 쉽게 가공할 수 있다.

이것에 대한 근본적인 원리를 파악하기 위해 다음과 같은 SQL을 실행시켜 보자

```
SELECT 성명, DECODE(구분, '작업', time) 작업, DECODE(구분, '휴식', time) 휴식,
       DECODE(구분, '중식', time) 중식, DECODE(구분, '교육', time) 교육,
       DECODE(구분, '회의', time) 회의, DECODE(구분, '석식', time) 석식,
       DECODE(구분, '특근', time) 특근, DECODE(구분, '야근', time) 야근
FROM 공수_테이블
WHERE 공수일자 = '19980723' and 작업반 = '밀링';
```

위의 SQL문 실행 결과는 각각의 로우마다 구분이라는 컬럼의 값에 의해 가공한 8개 컬럼중 하나의 컬럼에 정의된다. 또한 'WHERE' 조건에 해당되는 로우가 100건이면 처리한 결과의 로우도 100건이 된다. 이것을 테이블 형태로 표현하면 아래와 같은 형태가 된다.

성명	작업	휴식	중식	교육	회의	석식	특근	야근
홍길동	420							
홍길동		10						
홍길동			60					
홍길동		10						
차인표	380							
차인표		10						
차인표			60					
차인표				60				
차인표						60		
.	.	.	.	.	.	.	.	.

여기서 위와 같은 집합을 성명별로 한 로우씩 나타내고자 할 경우는 GROUP함수를 사용하면 된다.

```
SELECT 성명, SUM(DECODE(구분, '작업', time)) 작업,
       SUM(DECODE(구분, '휴식', time)) 휴식,
                    :
       SUM(DECODE(구분, '야근', time)) 야근
FROM 공수_테이블
WHERE 공수일자 = '19980723' and 작업반 = '밀링';
GROUP BY 성명
```

성명	작업	휴식	중식	교육	회의	석식	특근	야근
홍길동	420	20	60					
차인표	380	20	60	60		60	120	
김동훈		10				60		140
.	.	.	.	.	.	.	.	.

지금부터는 위와 같은 원리를 이용하여 활용범위를 확장시킨 사례에 대하여 알아보자.

1) 입력한 변수에 의해 일별, 월별로 집계하는 경우

입력 변수 형태 : 199807 -> 일별, 1998 -> 월별

```
SELECT SUBSTR(SDATE,1,LENGTH(:date)),
SUM(DECODE(형태||이자구분, 'S1', 0.1*잔액원금, 'S2', 0.2*할부원금)),
MAX(잔액원금), AVG(연체료), SUM(잔액원금)
FROM 입금_테이블
WHERE SDATE LIKE :date
GROUP BY SUBSTR(SDATE,1,LENGTH(:date));
```

2) 어떤 컬럼의 값에 의하여 집계하는 컬럼이 다른 경우

제품판매 구분이 국판일 때는 원화 금액을 사용하고 수출인 경우 외화금액에 환율을 적용하여 제품별로 판매금액을 추출한다.

```
SELECT PROD, SUM(DECODE(GUBUN, 'D', DAMT, 'S', SAMT*EXC))
FROM MACHULET
WHERE MDATE LIKE '1998007%'
GROUP BY PROD
```

3) 제품그룹별로 월, 당일 누계 등 다양한 집계를 구하는 경우

각각의 제품에 대해 제품그룹별 당일의 생산 수량과 해당 월의 주간별 누계 수량을 구하고 해당월의 누계 수량을 추출한 후 정렬시 특정 제품그룹으로 정렬한다.

```
SELECT DECODE(LENGTH(PROD),
'3',DECODE(PROD,'101','100','105','200',PROD),
'4',SUBSTR(PROD,1,3),
'5',DECODE(PROD,'105002','200',SUBSTR(PROD,1,3)))
SUM(DECODE(PDATE,TO_CHAR(SYSDATE),QTY)),
SUM(DECODE(TO_DATE(PDATE,YYYYMMDD,'W','1',QTY)),
                    :
SUM(DECODE(TO_DATE(PDATE,YYYYMMDD,'W','5',QTY)),
SUM(QTY) FROM PRESTMST
WHERE PDATE LIKE TO_CHAR(SYSDATE,'YYYYMM')||'%'
GROUP BY DECODE(LENGTH(PROD),
'3',DECODE(PROD,'101','100','105','200',PROD),
'4',SUBSTR(PROD,1,3),
'5',DECODE(PROD,'105002','200',SUBSTR(PROD,1,3)))
ORDER BY DECODE(SUBSTR(PROD,1,3),'520',PROD);
```

■ UNION ALL, UNION 등의 합집합을 활용한 사례

UNION ALL, UNION의 역할은 두개의 집합에서 비슷한 속성을 가진 컬럼들로 행으로 합쳐 하나의 새로운 합집합을 생성시킨다.

특히 GROUP BY와 UNION ALL을 함께 사용하면 새로운 열과 행으로 가공 처리할 수 있다.

위와 같은 3개의 테이블에서 특정일을 기준으로 제품에 대한 완료수량, 불량수량, 출고수량을 구하고자 한다. 이때 각 테이블에 해당 데이터가 존재할 수도 있고 존재하지 않을 수도 있다. 따라서 조인을 사용하여 원하는 데이터를 구할 수 없으므로 'UNION ALL' 을 사용하여 다음과 같은 SQL문으로 구할 수 있다.

```
SELECT 제품, MAX(DECODE(구분,1,QTY)) 완료, MAX(DECODE(구분,2,QTY)) 불량,
MAX(DECODE(구분,1,QTY)) 출고,
FROM ( SELECT 제품, '1', 구분 SUM(완료수량) QTY FROM 생산공정
WHERE 작업일자 = '19980725'
GROUP BY 제품
UNION ALL
SELECT 제품, '2', 구분 SUM(불량수량) QTY FROM 제품검사
WHERE 작업일자 = '19980725'
GROUP BY 제품
UNION ALL
SELECT 제품, '3', 구분 SUM(불량수량) QTY FROM 제품출고
WHERE 작업일자 = '19980725'
GROUP BY 제품 )
GROUP BY 제품
```

UNION, UNION ALL을 사용할 때는 가능한 UNION보다는 UNION ALL을 사용하는 것이 유리하며 비효율적인 처리요소가 발생되지 않도록 하는 것이 좋다.

INLINE VIEW는 여러개의 SQL문들을 하나로 통합할 수 있다

SQL의 활용범위를 한단계 향상시키는 길은 IN LINE VIEW를 최대한 잘 활용하는 것이라고 감히 말할 수 있다.

INLINE VIEW가 없을 때는 울며 겨자 먹기식으로 부득히 여러 개의 SQL를 사용하여 데이터를 처리할 수 밖에 없었다.

그러나 IN LINE VIEW를 활용하면 여러 개의 SQL을 사용하여 데이터를 처리하는 것을 하나의 SQL문으로 최적화할 수 있으며 수행속도를 획기적으로 개선할 수 있다고 생각한다.

이러한 이유에 대하여 다음의 사례를 통해 알아보자.

■ IN LINE VIEW를 통해 N개의 SQL을 하나로 통합

IN LINE VIEW를 사용하지 않고 좌측에 있는 테이블의 구조에서 우축의 데이터를 추출하기 위해서는 N개의 SQL를 사용해야만 가능하다.

성명	일자	부서	구분	내역
차인표	1995/01/20	510	1	대표이사상
차인표	1997/03/02	510	2	영어1급
차인표	1997/09/20	510	1	회장상
차인표	1996/11/30	510	3	기술사
차인표	1997/06/07	510	3	기사
차인표	1990/07/02	510	1	부사장 상
송승환	1998/02/01	510	1	회장상
.	.	.	.	.

(테이블)

부서 : 정보시스템실

성명	수 상	자격	가점
차인표	대표이사상	기술사	영어1급
	회장상	기사	
	부사장상		
송승환	대표이사상	기술사	영어2급
홍길동		기사	
.	.	.	.

(추출 데이터)

그러나 좌측 데이블이 아래와 같은 형태이면 쉽게 우측의 추출 데이터를 만들 수 있다.

성명	로우수	구분	내역
차인표	1	1	대표이사상
차인표	2	1	사장상
차인표	3	1	부사장상
차인표	1	2	영어 1급
차인표	1	3	기사
차인표	2	3	기술사
송승환	1	1	회장상

(가상 테이블)

따라서 기존의 테이블 형태를 가상의 테이블 형태로 만드는 것을 IN LINE VIEW로 처리하면 다음과 같은 SQL문을 만들 수 있다.

```
SELECT X.성명, X.RCNT, MAX(DECODE(구분,1,내역)),
MAX(DECODE(구분,2,내역)), MAX(DECODE(구분,3,내역))
FROM (SELECT 성명, ROWNUM RCNT, 구분, 내역 FROM TABLE 1
WHERE 부서 = '정보시스템' AND 구분 = '1'
UNION ALL
SELECT 성명, ROWNUM RCNT, 구분, 내역 FROM TABLE 1
WHERE 부서 = '정보시스템' AND 구분 = '2'
UNION ALL
SELECT 성명, ROWNUM RCNT, 구분, 내역 FROM TABLE 1
WHERE 부서 = '정보시스템' AND 구분 = '3' ) X
GROUP BY X.성명, X.RCNT;
```

■ IN LINE VIEW를 통한 조인 횟수 감소시켜 수행속도 향상

먼저 특정 테이블을 GROUP BY한 후 그 결과를 가지고 다른 테이블과 조인함으로써 조인 횟수를 감소시킬 수 있다. 예를 들어, 수주건별로 수주내역 테이블의 수주 금액과 입금실적 테이블의 입금액이 다른 로우를 추출한다면 대부분 다음과 같은 SLQ문을 사용한다.

```
SELECT X.수주번호, X.판매금액, SUM(Y.입금액)
FROM 수주테이블 X, 입금테이블 Y
WHERE X.수주번호 = Y.수주번호 AND X.수주일 LIKE '1998%'
GROUP BY X.수주번호, X.판매금액
HAVING X.판매금액 <> SUM(Y.입금액) ;
```

위의 SQL문에 대한 Trace 결과는 사용한 CPU Time은 62.12 초이고 Execution Plan은 다음과 같다.

```
Rows        Execution Plan
   0   SELECT  STATEMENT
 401    FILTER
35040     SORT  GROUP BY
1121280     NESTED LOOPS
35040         TABLE ACCESS  BY ROWID OF SUJUMAST
35040           INDEX  RANGE SCAN SUJUMAST_PK
1121280       TABLE ACCESS  BY ROWID OF INPRAS1T
1121280         INDEX  RANGE SCAN INPRAS1T_01X
```

여기서 문제점은 수주테이블과 입금테이블의 조인 횟수가 너무 많이 발생하는 것이다.

따라서 먼저 수주번호로 입금테이블을 GROUP BY 한 후 그 결과를 가지고 수주테이블과 조인하면 조인 횟수를 현저히 감소시킬 수 있으므로 다음과 같이 IN LINE VIEW를 활용하여 해결할 수 있다.

```
SELECT X.수주번호 ,X.판매금액,Y.입금액
FROM 수주테이블 X,(SELECT 수주번호,SUM(입금액) 입금액
                   FROM 입금테이블
WHERE 수주일 LIKE '1998%' GROUP BY 수주번호 ) Y
WHERE X.수주번호 = Y.수주번호 AND X.판매금액 <> Y.입금액;

Rows          Execution Plan
 401    FILTER
35040     NESTED LOOPS
35040       SORT  GROUP BY
1121280       INDEX  RANGE SCAN INPRAST1T_01X
35040     NDEX  RANGE SCAN SUJUMAST_PK
```

이상과 같이 IN LINE VIEW는 물리적으로 존재하지 않고 단지 논리적으로 하나의 독립적인 집합이다. 특히 SQL에 의하여 만들 수 있는 논리적인 집합이므로 이것을 잘 활용하면 보다 높은 수준의 SQL을 사용할 수 있다.

지금까지 SQL의 최적화 사례를 통해 살펴보았듯이 데이터베이스에서 제공하는 함수와 여러가지 기능들을 최대한 활용하면 SQL의 최적화 방법은 무한정으로 확장시킬 수 있다.