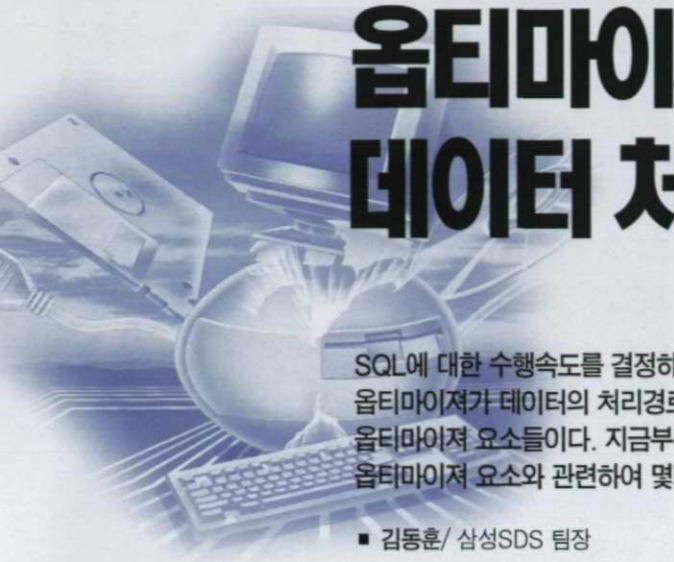
옵티마이져를 통한 데이터 처리경로의 최적화



SQL에 대한 수행속도를 결정하는 것은 옵티마이져가 작성한 데이터의 처리 경로이고 옵티마이져가 데이터의 처리경로를 작성하는데 가장 큰 영향을 주는 것은 사용자가 구성한 옵티마이져 요소들이다. 지금부터 데이터의 처리경로를 최적화할 수 있는 방법에 대하여 옵티마이져 요소와 관련하여 몇가지 설명하고자 한다.

■ 김동훈/ 삼성SDS 팀장

연 재 순 서

1. 수행속도 향상을 위한 기본 사항
2. 데이터 처리 경로를 최적화 - 이번호
3. SQL을 최적화

관계형 데이터베이스에서 모든 데이터의 처리 요구는 오르지 SQL을 통해서만 요구할 수 있으며, SQL은 단지 검색되어야 할 데이터의 집합을 정의하는 것 뿐이고 이 SQL에 대한 데이터의 처리 경로는 DBMS내의 옵티마이져가 옵티마이져 요소들을 참조하여 데이터의 처리경로(실행계획)를 수립한 후 실행한다.

따라서 SQL에 대한 수행속도를 결정하는 것은 옵티마이져가 작성한 데이터의 처리경로이고 옵티마이져가 데이터의 처리경로를 작성하는데 가장 큰 영향을 주는 것은 사용자가 구성한 옵티마이져 요소들이다.

일반적인 옵티마이져 요소들은 인덱스, 클러스터, 옵티마이져 모드, 데이터의 통계정보, 작성한 SQL의 문장, HINT 등이 있다. 특히 이러한 옵티마이져 요소들은 데이터의 처리경로를 결정하는데 한 가지만 영향을 미치는 것이 아니라 복합적으로 영향을 미친다.

그러므로 옵티마이져 요소들은 Application에서 요구하는 모든 형태의 데이터 처리에 활용될 수 있도록 구성해야 한다. 최적의 옵티마이져 요소들을 구성할 수 있는 능력은 SQL의 수행속도 향상 능력과 비례한다고 생각한다.

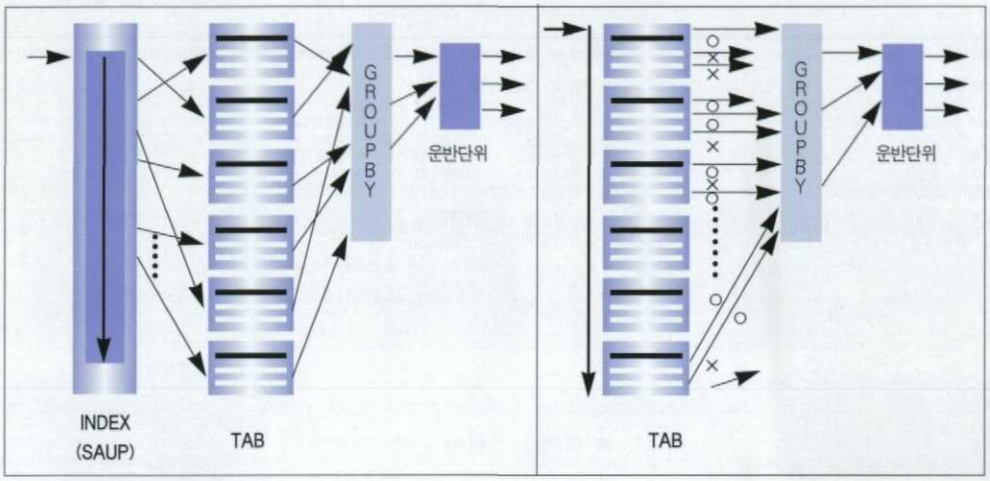
인덱스와 클러스터를 활용하여 데이터의 처리 경로 최적화

인덱스는 단지 옵티마이져가 최적의 데이터 처리 경로를 결정하기 위해 사용되는 하나의 오브젝트이며 우리가 일반적으로 생각하는 Key의 개념과는 다르게 이해해야 한다. 대부분의 사용자들은 인덱스를 사용하여 데이터를 검색하면 무조건 수행속도를 향상시킬 수 있다고 믿고 있다. 그러나 인덱스를 잘못 사용할 경우는 오히려 부하를 증가시켜 수행속도가 급격히 감소한다.

100,000건의 로우를 가진 테이블에서 80,000건의 데이터를 인덱스를 사용하여 추출할 때와 전체 테이블을 스캔하여 데이터를 추출한 경우를 비교하여 보자.

```
SELECT DEPT, SUM(QTY), SUM(AMT),SUM(STATUS,'20',QTY)
FROM PAY01TT
WHERE SAUP ='창원'
GROUP BY DEPT ;
```

인덱스를 사용할 경우는 좌측 그림과 같이 사업장명이 창원인 80,000 로우를 인덱스 스캔하면서 80,000번의 랜덤 액세스 방식으로 테이블에서 데이터를 추출한 후 부서별로 집계한다.



〈그림 1〉 인덱스 기본 개념

그러나 우측 그림과 같이 처리하면 테이블 전체를 스캔하면서 창원인 90,000건을 추출하여 부서별로 집계한다. 따라서 전체 테이블을 스캔하는 것이 90,000번의 랜덤 액세스를 발생하는 인덱스를 사용하는 것보다 훨씬 수행속도가 뛰어나다.

인덱스를 활용하여 데이터를 처리하고자 하는 경우 손익분기점 10~15% 이하이면 유리하고 이상이면 불리하다. 예를 들어 테이블의 전체 데이터가 100,000건인 경우 데이터의 처리범위가 15,000건 이하이면 인덱스를 사용하는 것이 유리하고 이상이면 전체 테이블을 스캔하는 것이 유리하다는 뜻이다.

인덱스를 사용하여 데이터를 처리하고자 할 때는 데이터의 처리 범위를 개발자가 미리 어느정도 파악한 후 사용해야만 한다. 테이블에 부서와 수주일자라는 컬럼으로 결합 인덱스가 생성되어 있고 다음과 같은 검색조건으로 데이터를 처리한다고 가정하여 보자

```
SELECT * FROM 수주테이블
WHERE 부서= :dept AND 일자 LIKE :yyyymm||'%'  AND 결재='Y';
```

이 SQL에서는 사용되는 인덱스의 테이터 처리범위는 아래의 SQL문을 사용하여 분석할 수 있다.

```
SELECT 부서,substr(수주일자,1,6), count(*) 처리 범위 from 수주테이블
GROUP BY 부서, substr(수주일자,1,6);
```

위의 SQL 결과에서 처리 범위의 값이 작을수록 수행속도는 뛰어나다. 특히 온라인으로 데이터를 처리할 때는 데이터의 처리 범위값을 작게 가지도록 인덱스를 구성하고 데이터의 처리 조건을 주어져야만 한다.

최적의 인덱스를 구성하여도 잘못된 SQL을 작성할 경우는 무용지물이 될 수 있으므로 인덱스의 사용규칙을 정확하게 이해하는 것은 매우 중요하다. 그리고 인덱스의 사용규칙을 적절하게 활용함으로써 옵티마이저가 판단하는 데이터의 처리 경로를 항상 사용자가 원하는 최적의 방향으로 유도할 수도 있다.

인덱스의 사용 규칙은 다음과 같이 정의할 수 있다.

1) 인덱스의 첫번째 컬럼을 SQL의 WHERE조건에 지정되어 있어야 한다.

2) 인덱스의 컬럼을 사용자가 함수나 연산자를 사용하여 임의적으로 변형하면 인덱스를 사용할 수 없다.

3) 인덱스의 컬럼에 대하여 서로 다른 데이터 타입을 비교하는 경우 DBMS내에서 임의적인 변형이 발생되어 인덱스를 사용할 수 없다.

4) 컬럼의 비교 연산자를 부정문으로 기술한 경우는 인덱스가 사용되지 않는다.

5) 인덱스의 컬럼이 NULL 값으로 비교되는 경우는 인덱스가 사용될 수 없다.

6) 옵티마이져 모드에 따라서 옵티마이져는 인덱스를 취사 선택하여 사용한다.

7) 사용자가 HINT를 사용하여 인덱스를 취사 선택할 수 있다.

인덱스를 활용하여 데이터의 처리 경로를 최적화한 사례를 살펴보자.

■ 인덱스만으로 처리

· 이 사례는 검색 데이터를 처리하는데 테이블의 데이터는 사용하지 않고 인덱스의 로우만으로 데이터를 처리함으로써 수행속도를 향상시킨 것이다.

· **인덱스 정보** : INPAMT_01X (INDATE,SUJUNO,IGUBUN)

· SQL :

```
SELECT SUJUNO, SUM(AMT) FROM INPAMT
WHERE INDATE LIKE '199803%' AND IGUBUN='1'
GROUP BY SUJUNO
```

· **Trace 결과** : CPU time -) 4초

Rows	Execution Plan
0	SELECT STATEMENT
10000	SORT GROUP BY
10000	TABLE ACCESS (FULL) OF 'INPAMT'
10001	INDEX (RNAGE SCAN) OF 'INPAMT_01X'

· **문제점** : 테이블의 로우에 대한 랜덤 액세스가 10000번 발생된다.

· **해결 방안** : AMT 컬럼을 인덱스에 추가한다면 인덱스만으로 데이터의 처리가 가능하기 때문에 랜덤 액세스를 제거할 수 있다.

INPAMT_01X (INDATE, SUJUNO, IGUBUN, AMT)

· **개선후 TRACE 결과** : CPU time -> 0.7 초

Rows	Execution Plan
0	SELECT STATEMENT
10000	SORT GROUP BY
10001	INDEX (RNAGE SCAN) OF 'INPAMT_01X'

■ 인덱스 구성이 잘못된 경우

· 이 사례는 데이터의 처리 범위를 감안하지 않고 인덱스를 구성함으로써 수행속도를 저하시킨 사례이다.

· **인덱스 정보** :

MDOUT0TT	MDOUT0TT_PK	REQDT, REQ_WHY, YETAKNO
	MDOUT0TT_01X	REQ_DT, REQ_WHY
	MDOUT0TT_02X	RESNO
MDOUT1TT	MDOUT1TT_PK	REQ_DT, REQ_WHY, RESNO, CHASU

· **SQL** :

```
SELECT y.yetakno, x.resno,x.req_why,
       sum(decode(x.chasu,'01',x.reqstk,0)),
       sum(decode(x.chasu,'01',x.reqamt,0)),
       sum(decode(x.chasu,'02',x.reqstk,0)),
       sum(decode(x.chasu,'02',x.reqamt,0)),
       sum(decode(x.chasu,'03',x.reqstk,0)),
       sum(decode(x.chasu,'02',x.reqamt,0))
FROM mdout1tt x, mdout0tt y
WHERE x.req_dt  = :a1    and x.req_why like :b1||'%'  and
      x.req_dt  = y.req_dt and x.req_why = y.req_why   and
      x.resno   = y.resno
GROUP BY y.yetakno,x.resno,x.req_why
```

· **Trace 결과** : CPU time -> 52.32초

Rows	Execution Plan
0	SELECT STATEMENT
2401	SORT GROUP BY
2401	NESTED LOOPS
2401	TABLE ACCESS BY ROWID OF MDOUT1TT
2405	INDEX RANGE SCAN OF MDOUT1TT_PK
2405	TABLE ACCESS BY ROWID OF MDOUT0TT
558900	INDEX RANGE SCAN MDOUT0TT_01X

· **문제점** : 인덱스 MDOUT0TT_01X를 사용한 데이터의 처리 범위은 넓은 반면 테이블에서 체크 기능을 하는 RESNO 컬럼의 데이터 범위는 매우 좁기 때문에 불필요한 랜덤 액세스량이 너무 많이 발생했다.

· **해결방안** : MDOUT0TT_IDX1의 인덱스 컬럼에 RESNO 컬럼을 추가함으로써 불필요한 랜덤 액세스량을 감소시킬 수 있다.

변경 후 : MDOUT0TT_IDX1 (REQ_DT,REQ_WHY,RESNO)

· **개선 후 Trace 결과** :

Rows	Execution Plan
0	SELECT STATEMENT
2401	SORT GROUP BY
2401	NESTED LOOPS
2405	TABLE ACCESS BY ROWID OF MDOUT1TT
2405	INDEX RANGE SCAN OF MDOUT1PK
2401	TABLE ACCESS BY ROWID OF MDOUT0TT
2401	INDEX RANGE SCAN MDOUT0TT_01X

■ 인덱스 컬럼의 순서

· 이 사례는 결합 인덱스에서 인덱스 컬럼의 순서가 수행 속도에 어떻게 영향을 미치는가를 보여준다.

· **인덱스 정보** :

PRODMST_PK (PROD)

SUJUMST_01X (DEPTCD + PART + SDATE)

OPTMAST_PK (OPTION)

· **SQL** :

```
SELECT X.PROD, X.OPTION, X.SDATE, X.QTY, Y.OPT_NAME,
       Z.PROD_NAME FROM SUJUMST X, OPTMAST Y, PRODMST Z
WHERE  X.OPTION = Y.OPTION  AND X.PROD  = Z.PROD AND
       X.SDATE  =  :b1      AND X.DEPTCD =  :b2  AND:
```

· **Trace 결과** :

CPU time -> 10 초

Rows	Execution Plan
0	SELECT STATEMENT
283	NESTED LOOPS
283	NESTED LOOPS
283	TABLE ACCESS (BY ROWID) OF 'SUJUMST'
57891	INDEX (RANGE SCAN) OF 'SUJUMST_01X'
283	TABLE ACCESS (BY ROWID) OF 'OPTMSST'
283	INDEX (UNIQUE SCAN) OF 'OPTMAST_PK'
283	TABLE ACCESS (BY ROWID) OF 'PRODMST'
283	INDEX (UNIQUE SCAN) OF 'PRODMST_PK'

· **문제점** : SQL의 실행순서를 보면 인덱스에 의한 정상적인 액세스가 이루어진 것처럼 보이지만 사실은 'SUJUMST' 테이블을 인덱스 'SUJUMST_01X'를 통하여 처리한 범위는 57891 ROW인 반면 테이블로 액세스한 횟수는 283이다.

· **해결방안** : 다른 애플리케이션에 영향이 없다면 'SUJUMST_01X' 인덱스 컬럼을 'DEPTCD + SDATE + PART' 또는 'SDATE + DEPTCD + PART'로 구성하면 두 컬럼이 모두 인덱스 처리조건으로 되어 인덱스 액세스 로우 수가 57891에서 283로 감소한다.

· **개선후 Trace 결과** : CPU time -> 0.6초

Rows	Execution Plan
0	SELECT STATEMENT
283	NESTED LOOPS
283	NESTED LOOPS
283	TABLE ACCESS (BY ROWID) OF 'SUJUMST'
283	INDEX (RANGE SCAN) OF 'SUJUMST_01X'

```
283      TABLE ACCESS ( BY ROWID ) OF 'OPTMSST'
283        INDEX ( UNIQUE SCAN ) OF 'OPTMAST_PK'
283      TABLE ACCESS ( BY ROWID ) OF 'PRODMST'
283        INDEX ( UNIQUE SCAN ) OF 'PRODMST_PK'
```

■ 인덱스를 활용하여 SORT 작업을 제거

· 이 사례는 WEB에서 가장 최근의 일자로 정렬하여 화면에 25건씩 데이터를 보여주는 경우인데 정렬작업으로 조건에 만족하는 모든 데이터를 추출한 후 결과를 추출함으로써 데이터량에 따라 급격히 수행속도가 저하되는 사례이다.

· **인덱스 정보** :

TOPIC1T_PK (TOPICNO), TOPIC1T_01X(TCHECK, TOPDATE)

· **SQL** :

```
SELECT * FROM TOPIC1T
WHERE TCHECK='Y'
ORDER BY TOPDATE DESC
```

· **Trace 결과** : CPU time -) 4초

```
Rows          Execution Plan
   0   SELECT STATEMENT
4000     SORT GROUP BY
4000       TABLE ACCESS ( BY ROWID ) OF 'TOPIC1T'
4000         INDEX (RNAGE SCAN) OF 'TOPIC1T_01X'
```

· **문제점** : 가장 최근 일자별로 데이터를 25건만 추출하여 화면에 보여준 후 사용자가 추가적인 데이터 요구시 다음의 25건만 보여주면 된다. 그러나 위의 SQL을 사용하면 조건에 만족하는 모든 데이터를 검색하여 최근의 일자별로 정렬한 후 데이터를 추출한다. 이런 경우 조건에 만족하는 로우의 수에 의해 수행속도는 급격히 감소한다.

· **해결 방안** : 정렬 작업만 제거하기 위하여 인덱스 TOPIC1T_01X을 역순으로 스캔한 후 부분 처리 방식으로 25건만 먼저 추출하여 화면에 보여주면 된다. 이 경우 테이블의 데이터량에 관계 없이 항상 일정한 수행 속도를 유지한다

```
SELECT --+ INDEX_DESC(TOPIC1T TPOIC1T_01X)
         * FROM TOPIC1T
WHERE TCHECK='Y';
```

· **개선후 TRACE 결과** : CPU time -) 0.2 초

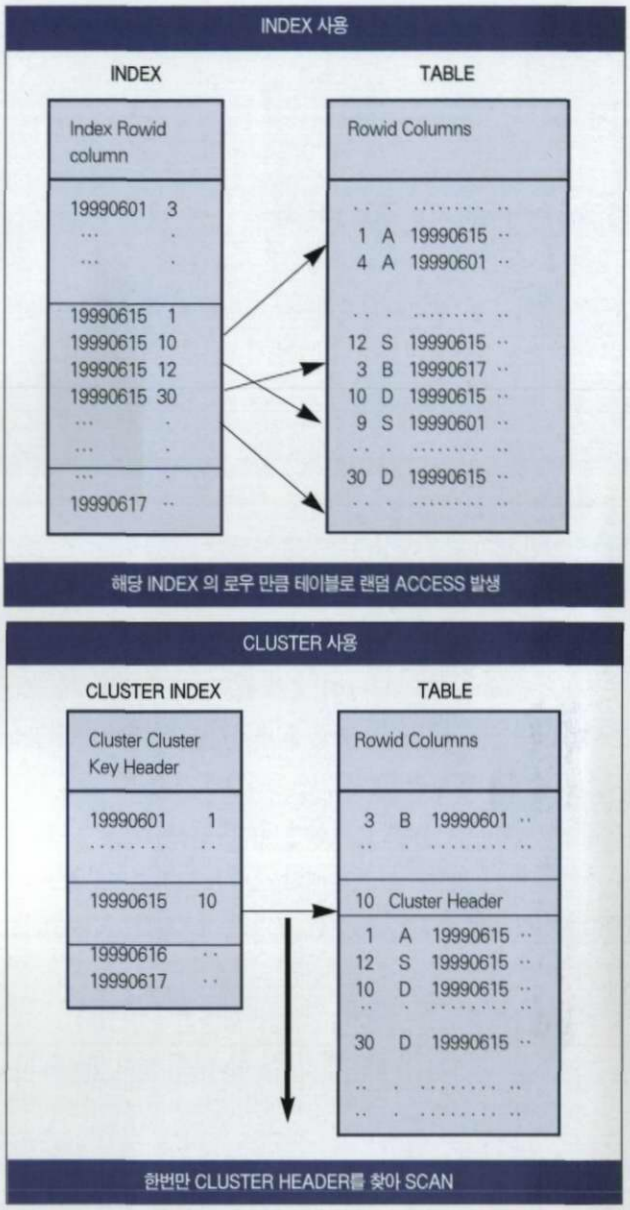
```
Rows          Execution Plan
   0   SELECT STATEMENT
  25     TABLE ACCESS ( BY ROWID ) OF 'TOPIC1T'
  26       INDEX (RNAGE SCAN) OF 'TOPIC1T_01X'
```

클러스터는 특정 테이블에서 특정 컬럼에 대하여 동일한 값을 가진 모든 로우를 같은 장소에 저장하는 것을 말한다. 또한 일반적인 테이블과 거의 동일하며 단지 인덱스와 다르게 랜덤 액세스를 감소시키는 요소만 추가된 형태이다.

특히 단일 테이블 클러스터는 한번의 클러스터 인덱스를 액세스하여 테이블의 로우들을 스캔방식으로 처리함으로써 인덱스의 약점인 랜덤 액세스를 줄여주는 역할을 한다.

이것을 인덱스와 비교하여 표현하면 〈그림 2〉와 같다.

아래의 SQL문을 인덱스를 활용할 경우 온라인으로 처리하기



〈그림 2〉 인덱스와 클러스터 비교

는 수행속도에 약간의 부담이 발생된다. 그러나 클러스터를 활용하면 액세스 효율을 극대화할 수 있어 온라인으로 처리시 수행속도에 부담이 없다.

```
SELECT SUM(SALEQTY) FROM SUJUMAST
WHERE SUJUDATE LIKE '199906%'
```

```
인덱스 활용시 : 1초
2500 SORT GROUP BY
2500 TABLE ACCESS BY ROWID 'SUJUMST'
2501 INDEX RNAGE SCAN 'SUJUMST_01X'
클러스터 활용시 : 0.2초
2500 SORT GROUP BY
2500 TABLE ACCESS CLUSTER 'SUJUMST'
15 INDEX RNAGE SCAN 'SUJUMST#'
```

이상의 사례에서 보는 것과 같이 인덱스와 클러스터를 적절하게 구성하고 활용하는 경우 사용자가 데이터의 처리경로를 최적화할 수 있다

조인시 데이터의 처리 경로 최적화

관계형 데이터베이스의 최대 장점은 보다 편리하게 관련된 정보를 연결하여 다양한 형태의 데이터 처리를 할 수 있다는 것이다. 그러므로 조인은 관계형 데이터베이스에서 가장 기본적이고 중요한 기능이라고 생각한다.

관계형 데이터베이스 이전의 데이터베이스에서는 원하는 정보를 연결할 수 있도록 하기 위해서는 어떤 물리적인 연결고리를 만들어야만 했기 때문에 설계시에 많은 노력이 필요했다. 왜냐하면 시스템 개발시 예상하지 못한 비즈니스 프로세스가 발생하여 데이터베이스의 설계 변경작업을 하는 경우 시스템의 거의 모든 부분을 수정해야 하는 어려움이 있기 때문이다. 그러나 이러한 문제들을 쉽게 개선할 수 있도록 한 것이 바로 관계형 데이터베이스이다. 관계형 데이터베이스는 논리적인 연결고리만 있으면 자유롭게 원하는 데이터를 연결하여 데이터를 처리할 수 있다.

조인을 할 때 최적의 데이터 처리 경로를 찾고 관계형 데이터베이스의 장점을 최대한 활용하여 SQL을 최적화하는 경우 기존의 데이터베이스에서 다른 데이터를 연결하여 검색하는 것과 거의 비슷한 수행속도 또는 더 빠른 수행속도를 얻을 수 있다.

조인시 수행속도를 결정하는 요소는 데이터의 처리 범위에 따른 조인의 순서, 조인의 연결관계를 가지는 컬럼의 인덱스 또는 클러스터 존재 여부, 세개 이상의 조인시 첫번째 조인의 성공률, 조인의 방법 등이 있다. 따라서 개발자는 이러한 원리에 맞도록 최적의 데이터 처리경로를 이미 판단할 수 있어야 하고 옵티마이져가 이것대로 수행할 수 있도록 옵티마이져 요소를 활용할 수 있어야 한다.

다음은 조인시 수행속도를 결정하는 요소에 대하여 자세히 살펴보자.

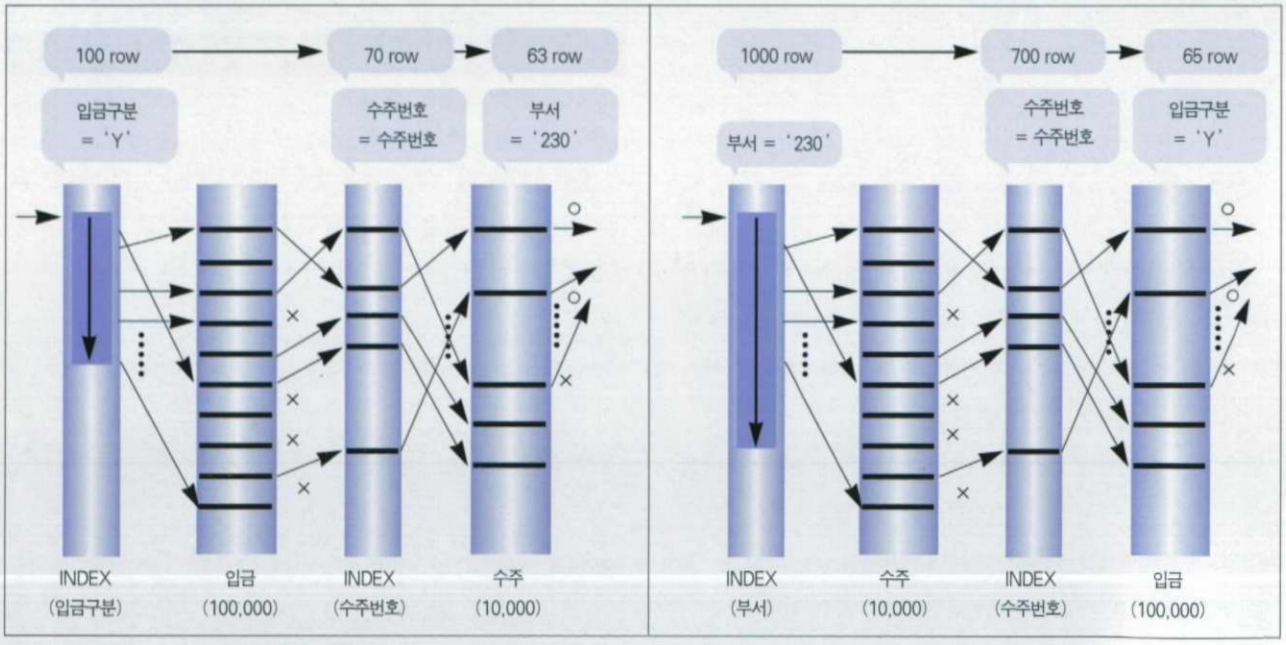
첫째, 데이터의 처리범위가 적은 데이블부터 먼저 수행하도록 한다.

```
SELECT * FROM 수주 X, 입금 Y
WHERE X.수주 = Y.입금 AND
        X.부서 = '230' AND Y.입금구분 = '4'
```

〈그림 3〉에서 보는 것과 같이 데이터의 처리 경로는 서로 다르지만 결과는 동일하다. 그러나 우측 그림은 입금 테이블의 입금구분 컬럼에 있는 인덱스를 사용하여 처리할 데이터의 처리범위는 100건이므로 성공하든 실패하든간에 조인 횟수는 100회 발생한다.

좌측 그림에서는 수주 테이블의 부서 컬럼에 있는 인덱스를 사용하여 처리할 데이터의 처리범위는 1,000건이기 때문에 조인은 1,000회 발생한다. 따라서 조인의 횟수는 먼저 처리하는 테이블의 데이터 처리범위에 의하여 결정되어 진다.

그러므로 'WHERE' 조건에 의하여 정해진 데이터의 처리범



〈 그림 3〉 데이터의 처리 범위에 따른 조인의 순서

위들중에서 가장 작은 처리범위를 가진 테이블이 먼저 수행될수 있도록 옵티마이져 요소를 적절하게 지정하거나 힌트나 인덱스를 활용하면 최적의 데이터 처리경로를 가질 수 있는 조인을 구현할 수 있다.

위 예의 SQL에서 우측그림과 같은 처리경로를 가지도록 유도하는 간단한 방법은 다음과 같다.

```
SELECT * FROM 수주 X, 입금 Y
WHERE X.수주 = Y.입금 AND
        RTRIM(X.부서) = '230'  AND Y.입금구분 = '4'
```

```
SELECT    A.PROD_NAME, SUM (A.SALE_AMT * A.UNIT_PRICE)
  FROM    SALE A,  PRODUCT B
WHERE     B.PROD_NO BETWEEN 'P20150' AND 'P20200'
   AND    A.PROD_NO = B.PROD_NO
   AND    A.SALEDATE LIKE '1995%'
GROUP     BY A.PROD_NO ;

Rows          Execution Plan
   0   SELECT  STATEMENT
10032    SORT GROUP BY
10032      NESTED  LOOPS
   20       TABLE  ACCESS ( BY ROWID ) OF 'PRODUCT'
   21         INDEX ( RANGE SCAN ) OF 'PRODUCT_PK'
10032       TABLE  ACCESS ( BY ROWID ) OF 'SALE'
15283         INDEX ( UNIQUE SCAN ) OF 'PRODNO_01X'
```

둘째, 세개 이상의 테이블을 조인할 때는 조인에 성공한 로우수가 적은 것부터 먼저 연결 작업이 수행되도록 한다.

〈그림 4〉에서 좌측 그림과 같이 데이터의 처리 경로를 사용할 경우 첫번째 조인한 횟수는 100회이고 조인을 성공한 결과는 50건이다. 따라서 두번째 조인횟수는 50회이므로 총 조인의 횟수는 150회가 된다.

우측 그림과 같은 데이터 처리경로를 사용할 경우 첫번째 조인한 횟수는 100회이고 조인 100건이므로 두번째 조인횟수는 100회이다. 따라서 총 조인의 횟수는 200회가 된다. 이상과 같이 먼저 조인한 결과의 로우는 다음에 조인할 데이터의 처리범위가 되기 때문에 조인을 성공한 로우의 수가 작을수록 다음에 조인할 조인 횟수는 감소한다.

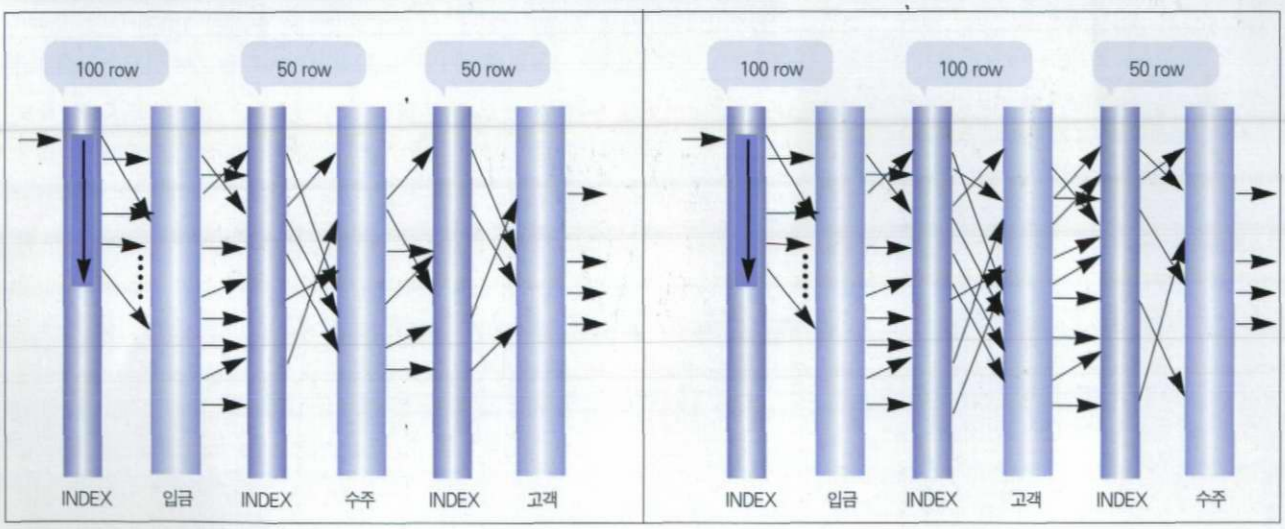
이러한 원리를 근거로 조인시 데이터의 처리경로를 최적화한 사례를 알아보자.

다음 사례는 1:M의 관계를 가진 두개의 테이블을 조인하여 특정값으로 집계하는 경우 1:M 관계를 1:1관계로 변환한 후 집계함으로써 조인의 횟수를 감소시킨 사례이다.

이 SQL문을 실행시 문제점은 먼저 조인을 한 후 제품별로 집계됨으로써 불필요하게 많은 조인 횟수가 발생된다. 따라서 조인 횟수를 감소시킬 수 있도록 주어진 해당월에 대해 SALE 테이블을 제품별로 집계를 한 후 PRODUCT 테이블에 조인할 수 있도록 데이터의 처리 경로를 유도하면 항상 제품 건수만큼만 조인 횟수가 발생하게 된다.

지금까지 살펴본 내용은 데이터의 처리경로를 최적화하는 기본적인 방법과 사례들을 개략적으로 설명하였다.

이 부분에 대한 모든 내용을 설명하려면 이 지면을 모두 사용하여도 설명하기 힘들 정도로 많은 내용이다. 그러나 이 글에서 설명한 내용과 같이 인덱스를 구성하고 데이터의 처리경로를 최적화할 경우 보다 향상된 시스템을 구축할 수 있다고 생각한다.



〈그림 4〉 3개 이상의 테이블 조인인 경우