수행속도 향상시키는 지름길

대부분의 정보시스템 개발 종사자들은 대단위 관계형 데이터베이스의 정확한 개념 정립이 되어 있지 않은 상태에서 시스템 개발에 접근한다. 이것은 많은 비효율을 발생시켜 시스템 개발 생산성 및 품질의 저하를 가져오며, 시스템의 수행속도를 현저히 떨어뜨린다. 특히, 관계형 데이터베이스는 활용 능력에 따라 효율의 차이가 매우 크게 나타나며, 이는 바로 프로젝트의 성공과 실패를 판가름하게 된다. 이 글에서는 관계형 데이터베이스를 보다 효율적으로 활용할 수 있는 다양한 방법과 실무에서 개발자 일량을 감소시키면서 수행속도를 향상시키는 솔루션을 제공하고자 한다.

김동훈/삼성 SDS팀장

연 · 재 · 순 · 서

1회 : 수행속도 향상을 위한 기본 사항
2회 : 데이터 처리 경로를 최적화
3회 : SQL을 최적화

관계형 데이터베이스에서 데이터의 처리는 하나 또는 여러 개의 테이블(집합)에서 사용자가 원하는 데이터(집합)를 찾아서 원하는 형태(집합)로 만드는 것이다. 여기서 사용자가 원하는 데이터를 찾는 것과 원하는 형태로 만드는 것을 어떻게 효율적으로 최적화 하느냐가 관계형 데이터베이스의 수행속도를 결정한다고 생각한다.

따라서 데이터의 처리를 최적화할 수 있는 다양한 방법은 다음호에 구체적으로 설명하기로 하고, 먼저 수행속도 향상을 위한 기본 사항에 대하여 알아보기로 하겠다.

옵티마이저가 최적의 실행계획을 수립하도록 유도한다.

관계형 데이터베이스에서는 사용자가 SQL을 이용하여 데이터의 처리를 요구하면 DBMS내에 있는 옵티마이저가 자료사전(Data Dictionary)을 참조하여 SQL을 해석하고,데이터 처리를 위한 실행계획(데이터의 처리 경로)을 수립한 후 처리한다. 이 과정을 파싱(Parsing)이라고 부르며 이 실행 계획에 따라서 수행속도는 결정된다.

옵티마이저가 실행 계획을 수립하는데 큰 영향을 주는 요소들은 인덱스와 클러스터, 옵티마이저 모드, ANALYZE 실행으로 생성된 통계 정보, 실행한 SQL의 문장, DBMS 버전에 따른 옵티마이저의 차이 등 많은 요소들이 있다.

실행계획 수립시 이런 요소들 중에서 한가지만 영향을 주는 것이 아니라 여러 가지 요소들이 복합적으로 영향을 미친다. 따라서 관계형 데이터베이스를 오랫동안 사용한 분들은 관계형 데이터베이스는 배우면 배울수록 어렵고 이해할 수 없다라고 말한다. 이것은 실행계획에 영향을 주는 요소들의 정확한 원리 및 개념을 이해하지 못하기 때문이다.

옵티마이저는 동일한 SQL이라 하더라도 비교한 컬럼의 데이터분포도와 통계 정보의 생성 주기, 인덱스의 생성 시점 등의 차이에 따라 실행 계획이 아주 다르게 나타날 수 있기 때문에 사용자들은 많은 혼란을 가지게 된다.

그러나 옵티마이저 요소들을 적절하게 활용함으로써 옵티마이저가 수립하는 실행 계획을 항상 사용자가 원하는 최적

의 방향으로 유도할 수 있다. 따라서 SQL을 최적화 하기 위해서는 먼저 데이터 처리에 대한 최적의 실행계획을 수립할 수 있는 능력이 있어야 하고, 이 실행 계획을 옵티마이저가 수립할 수 있도록 옵티마이저 요소들을 생성하고, SQL을 비절차형으로 기술해야만 수행속도를 향상시킬 수 있다.

또한 최적화된 SQL은 약간의 옵티마이저 요소들을 적절히 조정하거나, SQL을 조금만 변형할 경우 쉽게 최적화된 실행계획으로 만들 수 있다. 그리고 데이터의 처리과정에서 발생된 문제점을 쉽게 찾을 수 있고, 간단한 튜닝 작업만으로도 수행속도를 수십 또는 수백배로 향상시킬 수 있다.

옵티마이저가 최적의 실행계획을 수립할 수 있도록 사용자가 유도하는 방법을 간단히 소개하면 다음과 같다.

SQL> SELECT sujuno, sujudate FROM sujumst WHERE custno= 'D00234' AND gubun = '2'	TABLE ACCESS BY ROWID SUJUMST AND-EQUAL INDEX RANGE SCAN SUJU_CUSTNO INDEX RANGE SCAN SUJU_GUBUN

SQL> SELECT sujuno, sujudate FROM sujumst WHERE custno= 'D00234' AND RTRIM(GUBUN) = '2'	TABLE ACCESS BY ROWID SUJUMST INDEX RANGN SCAN SUJU- CUSTNO

〈인덱스 정보〉

- ibsomst : ibsomst_pk(ibso_no), ibsomst_01x(ibso_date, jaso_yn)
- hyungmst : hyungmst_pk (ibso_no, hyung_gubun)
  hyungmst_01x(gigwan_cd, jaso_yn, sinbun_cd, call_no)

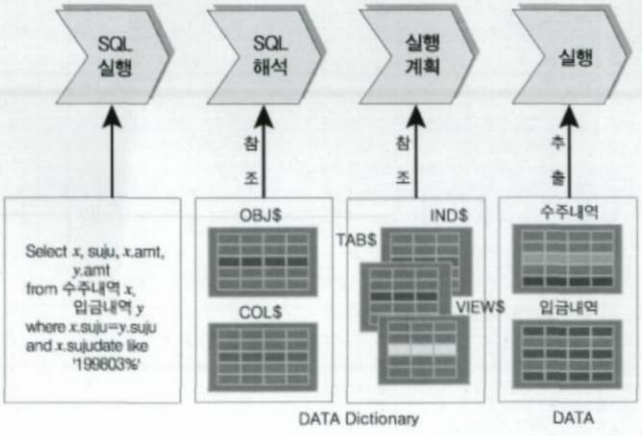
〈 옵티마이저의 데이터 처리 경로 〉

```
SELECT x.gigwan_cd, x.ibso_no, x.ibso_date, y.hyung
FROM ibsomst y hyungmst x
WHEREx.ibso_no=y.ibso_no and x.jaso_yn = 'Y' and x.ibso_date= '199990510'
andy.gigwan_cd=x.gigwan_cd and y.jaso_yn=x.jaso_yn andy.sinbun_cd='1' ;
ROWS EXECUTIONPLAN
0 SELECT STATEMENT
167 NESTED LOOPS
167TABLEACCESS ( BY ROWID )OF 'IBSOMST'
168 INDEX( RANGE SCAN ) OF 'IBSOMST_01X'
142 TABLEACCESS ( BY ROWID )OF 'HYUNGMST'
57238INDEX( RANGE SCAN ) OF 'HYUNGMST_01X'
```

〈 사용자가 유도한 데이터 처리 경로 〉

```
SELECT x.gigwan_cd, x.ibso_no, x.ibso_date, y.hyung
FROM ibsomst y hyungmst x
WHEREx.ibso_no=y.ibso_no and
x.jaso_yn = 'Y' and x.ibso_date= '199990510' and
rtrim(y.gigwan_cd)= x.gigwan_cd and y.jaso_yn=x.jaso_yn and
y.sinbun_cd= '1' ;
ROWS EXECUTIONPLAN
0 SELECT STATEMENT
168 NESTED LOOPS
167TABLEACCESS ( BY ROWID )OF ' IBSOMST'
168 INDEX( RANGE SCAN ) OF ' IBSOMST_01X'
142 TABLEACCESS ( BY ROWID )OF ' HYUNGMST'
168INDEX( RANGE SCAN ) OF 'HYUNGMST_PK'
```

소개된 내용과 같이 사용자가 간단하게 데이터의 처리 경로를 변경시킬 수 있다. 이것은 옵티마이저가 실행계획을 수립할 때 〈그림 1〉에서 보는 것처럼 사용된 테이블의 옵티마이저 요소들을 참조하여 실행계획을 수립하기 때문이다.



〈그림 1〉 SQL의 수행단계

SQL을 하나의 Application 단위로 활용한다.

아래의 두가지 예를 통해 데이터의 처리 결과는 동일하지

만 여러개의 SQL문을 사용하여 처리하는 경우와 하나의 SQL문으로 처리하는 경우의 수행속도 차이를 서로 비교하여 보자

〈예제 1〉 SQL을 반복 수행하여 데이터를 처리

```
DECLARE
sujunovarchar2(12);
sujudatevarchar2(08);
igubunnumber;
iamtnumber(10, 3);
tamtnumber(10, 3);
CURSOR c1 IS
select sujuno, sujudate, igubun, iamtm
from psc324
where sujudate like '199905%';
BEGIN
OPEN c1;
LOOP
FETCH c1 INTO :sujuno, :sujudate, :igubun, :iamt;
EXIT WHEN c1%NOTFOUND;
IF IGUBUN<=10 THEN
tamt := iamt * 0.5
ELSE
tamt := iamt * 0.3
END IF;
insert into psc325
values ( :sujuno, '19990601', igubun, :iamt, :tamt );
COMMIT;
END LOOP;
CLOSE C1;
END;
```

〈예제 1〉의 실행결과를 Tarce정보로 자세히 분석하여 보면

```
SELECT SUJUNO, SUJUDATE, IGUBUN, IAMTM FROM PSC324
WHERE SUJUDATE LIKE '199905%';

 CALL   COUNT   CPU   ELAPSED   DISK   QUERY CURRENT ROWS
------- ------ ----- --------- ------ ----- ------- ------
 PARSE      1   0.01      0.02      0     0       0      0
EXECUTE    64   1.04      2.34      0     0       0      0
 FETCH   1536   6.82     10.74      0  7242       0   1472

ROWS EXECUTIONPLAN
-------------------------------------------------------
```

```
  0  SELECT STATEMENT
3584  TABLEACCESS ( BY ROWID )OF ' PSC324 '
3776  INDEX( RANGE SCAN )OF ' PSC324_X1 '

INSERT INTO PSC325
VALUES ( :SUJUNO, '19990601', IGUBUN, :IAMT, :TAMT );
 CALL   COUNT   CPU   ELAPSED   DISK   QUERY CURRENT ROWS
------- ------ ----- --------- ------ ----- ------- ------
 PARSE      1   0.08      0.11      0     0       0      0
EXECUTE  1472   6.88     10.91    371  5752   27136   1472
 FETCH      0   0.00      0.00      0     0       0      0

ROWS      EXECUTION          PLAN
--------- ------------------------------------------------
0         INSERT             STATEMENT

COMMIT;

 CALL   COUNT   CPU   ELAPSED   DISK   QUERY CURRENT ROWS
------- ------ ----- --------- ------ ----- ------- ------
 PARSE      1   0.08      0.11      0     0       0      0
EXECUTE  1472   5.18      8.91    371  5752   27136   1472
 FETCH      0   0.00      0.00      0     0       0      0
```

〈예제2〉 하나의 SQL로 데이터를 처리

```
insert into psc325
(select sujuno, '19950601', igubun, iamt,
decode(sign(10-igubun, -1, iamt*0.3, iamt*0.5)
from psc324
wheresujudate like '199905%' ) ;
COMMIT;
```

〈예제 2〉의 실행결과를 Trace 정보로 분석하면

```
 CALL   COUNT   CPU   ELAPSED   DISK   QUERY CURRENT ROWS
------- ------ ----- --------- ------ ----- ------- ------
 PARSE      1   0.08      0.11      0     0       0      0
EXECUTE     1   2.01      2.25    371  5752   27136   1472
 FETCH      0   0.00      0.00      0     0       0      0

ROWS EXECUTIONPLAN
--------- ------------------------------------------------
0         INSERT     STATEMENT
3584  TABLEACCESS ( BY ROWID )OF 'PSC324'
3776      INDEX( RANGE SCAN ) OF 'PSC324_X1'
```

두가지 예제에서 보는 것과 같이 처리한 결과는 동일하지

만 데이터를 처리하는 방법에 따라 수행속도는 약 9배 정도의 차이를 보이고 있다.

일반적으로 Application이 수행될 때 SQL을 만나게 되면 데이베이스 호출(DBMS CALL)을 하게 되는데 이것이 많이 발생되면 DBMS 오버헤드가 발생된다. 〈예제1〉의 경우 Loop안에 있는 SQL문들은 커서에서 Fetch한 로우의 수만큼 DBMS CALL을 수행한다. 따라서 〈예제2〉와 같이 한번의 DBMS CALL을 통해 데이터를 처리하는 것이 〈예제1〉 경우보다 시스템의 부하를 감소 시키고 훨씬 빠른 수행속도를 보장받을 수 있다.

또한 〈예제1〉에서 보관 커서를 사용하더라도 수행속도는 많은 차이가 발생한다. 왜냐하면 〈예제 2〉는 한번의 내부 커서(Implicit Cursor)로 모든 처리가 완료 되지만, 〈예제 1〉은 여러번의 외부커서를 생성해야 하고, 보관 커서가 파싱의 전체단계를 감소시켜 주는 것이 아니기 때문이다.

컨설팅 업무를 하다보면 개발자들은 〈예제2〉와 같은 방법으로 데이터를 처리하는 경우를 흔히 볼 수 있다. 이것은 관계형 데이터베이스의 정확한 개념을 이해하지 않고 기존의 3GL 코딩 방식을 그대로 답습하고 있기 때문이다.(C/S 환경에서 데이터 처리의 최적화는 더욱 필요하다).

현재 관계형 데이터베이스를 적용하여 개발되는 대부분의 시스템은 C/S 환경을 채택하고 있는데, 이러한 환경에서 데이터 처리의 최적화는 더욱더 중요하다고 생각된다. 왜냐하면 2-Tier 또는 3-Tier 구조에서도 수행 속도를 결정하는 것은 결국 서버에서 수행되는 데이터 처리 작업이기 때문이다.

만일 C/S Application에서 월별로 평균 5만 건이 발생되는 생산현황 테이블에서 제품별로 당월과 당일의 계획 대비 실적 현황을 보고자 한다.

〈표〉 국내 암호화 제품 자체개발업체

제 품 명	월 누계 실적(1998/03)			당일 실적(1998/03/21)		
	계획	계획	달성률(%)	계획	실적	달성률(%)
고급	1210	1120	93	160	172	117
중급	720	680	94	80	750	75
보급용	2800	2720	90	130	130	100
전문가용	500	530	106			
특수용	340	250	74	20	20	100

만일 당월분의 데이터 5만 건을 클라이언트로 내려받은 후 로직으로 처리하고, 동일한 방법으로 당일분까지 처리한다고 가정하여 보자. 이런 경우 다량의 데이터가 네트워크를 타고 클라이언트로 이동하기 때문에 네트워크 트래픽이 크게 증가하게 되고 상대적으로 처리속도가 늦은 클라이언트에서 다량의 데이터를 원하는 모양으로 가공한다면 최종 사용자가 느끼는 수행 속도는 좋을 수가 없다. 또한 당일의 계획과 실적이 없는 경우 제품별로 당일분의 데이터가 틀리게 된다.

이것을 다음과 같이 최적화하면 개발생산성 향상과 위에서 발생되는 모든 문제들을 자동적으로 해결할 수 있다.

```
SELECT 제품명, 당월계획, 당월실적, 당월달성률, 당일계획, 당일실적, 당일달성률
FROM (SELECT decode(length(prod),4, substr(prod,1,3),5, substr(prod, 1, 4), prod) PROD, sum(decode(gubun, '계획' ,qty)) 당월계획,sum(decode(gubun, '실적' ,qty)) 당월실적, sum(decode(gubun, '실적' ,qty))/sum(decode(gubun, '계획' ,qty))*100 당월달성률(%), sum (decode(gubun||pdate, '계획' ||:sdate,qty)) 당일계획, sum (decode(gubun||pdate, '실적' ||:sdate,qty)) 당일실적, (sum (decode(gubun||pdate, '실적' ||:sdate,qty)) /sum(decode (gubun||pdate, '계획' ||:sdate,qty)))*100 당일달성률(%)
FROM PRODUCT
WHEREPDATE LIKE SUBSTR(:pdate,1,6)|| '%'
GROUP BY decode(length(prod), 4, substr(prod, 1, 3), 5, substr (prod, 1, 4), prod) ) X,
(SELECT prod_code, prod_name 제품명 FROM PROD_CODE ) Y
WHERE Y.PROD_CODE =X.PROD
```

이유는 서버에서 제품별로 GROUP BY를 하였기 때문에 5건의 데이터량만 네트워크를 통해 클라이언트로 이동하므로 네트워크 트래픽은 거의 없다. 또한 서버에서 모든 데이터의 처리와 가공을 완료하였기 때문에 수행속도 향상은 물론 클라이언트에서 특별히 로직 처리할 일이 거의 없기 때문에 개발 생산성이 크게 향상된다.

다양한 데이터의 연결과 함수를 활용한다.

SQL은 데이터의 처리 과정에서 얻어진 값을 다른 처리에 사용할 수 없고 다만 처리한 결과를 얻을 뿐이다. 따라서 일

반적으로 데이터의 건별로 처리 과정을 다르게 하고자 할때는 먼저 SQL을 통하여 데이터를 검색한 후 'IF' 및 'CASE' 등을 사용하여 분기한 후 다시 SQL을 통하여 데이터를 처리한다. 그러나 이것은 사용자의 데이터 처리 로직이 수반되어 SQL의 활용도를 감소시키고 전체의 처리경로 최적화를 어렵게 한다.

이런 경우는 SQL의 활용 폭을 넓히기 위해 조인을 활용하거나 SQL함수를 활용하는 것이 유리하다. 조인을 활용하면 여러 개의 SQL을 하나의 SQL로 통합할 수 있고 처리과정 중에 있는 서로의 값을 이용하여 매우 복잡한 데이터를 처리할 수 있기 때문이다.

이때 조인을 사용함으로써 수행 속도가 저하된다고 생각할 수 있으나 조인을 사용한다고 해서 무조건 수행 속도가 저하된다는 생각은 잘못된 생각이다. 특이한 몇가지 경우를 제외하고는 여러개의 SQL을 사용하거나 SQL을 반복 수행하는 것보다 조인을 활용하는 것이 훨씬 수행속도를 증가시킬 수 있다. 조인의 수행 속도에 큰 영향을 주는 원리와 처리경로를 최적화할 수 있는 자세한 방법은 다음에 설명하기로 하겠다.

그 다음 조인을 활용하여 통합한 정보를 다양한 경우의 수를 처리할 수 있도록 SQL함수들을 활용한다.(SQL함수들은 관계형 데이터베이스 각 업체별로 다를 수 있다)

SQL내에서 'IF' 와 같이 분기하여 데이터를 처리할 수 있는 방법은 'DECODE' 를 활용하는 방법과 '사용자 정의 저장형 함수(User Defined Stored Function)' 를 활용하는 방법 등 크게 두가지 활용 방법이 있다.

여기서 'DECODE' 는 특정 업체에서만 사용할 수 있다는 제한이 있다. '사용자 정의 저장형 함수(User Defined Stored Function)' 를 활용하는 방법은 데이터베이스가 보유하고 있는 절차형 SQL 언어를 이용하여 사전에 저장형 함수를 생성하고 이를 SQL 내에서 사용한다.

아래와 같이 1 : M의 관계가 있는 수강과목별 정원 테이블과 학생별 수강과목 테이블을 서로 비교하여 수강 신청을 할 때 수강과목의 정원을 초과할 수 없도록 체크하는 SQL 문을 작성하여 보자

1 : M

수강과목별 정원관리 (SUBJECT-CNT)	학생별 수강과목 (SUBJECT)

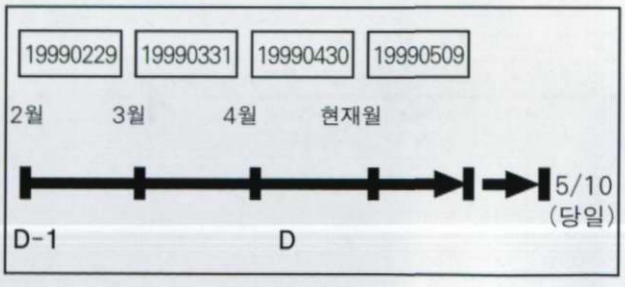
```
CREATE OR REPLACE FUNCTION subject_cnt(i_subject_noin
varchar2)
return number is
subject_cnt number ;
BEGIN
SELECT COUNT(*) INTO :SUBJECT_CNTFROM SUBJECT
WHERE IGUBUN IN ( 'A' , 'B' ) AND SUBJECT_NO=:I_SUBJECT_NO ;
RETURN subject_cnt;
END subject_cnt;

SELECT STUDENT_NO, ADDR, NAME .....
FROM ( SELECT x.STUDENT_NO, x.ADDR, x.NAME,
SUB_CNT, subject_cnt(X.SUBJECT_NO) as SUBJECT_CNT,
FROM STUDENT x, SUBJECT_CNT z
WHERE x.SUBJECT_NO = z.SUBJECT_NO
AND x.STUDENT_NO = '99072220' )
WHERE subject_cnt < sub_cnt;
```

전일(D-1일)까지의 모든 데이터 + 당일(D) 데이터 정보는 실시간 정보이다.

일반적으로 경영자 정보에서는 통계성 데이터를 Real Time으로 검색하기를 원한다. 따라서 보통 벌크 데이터를 직접 집계하여 정보를 제공하든가 아니면 배치 작업으로 벌크 데이터를 전일까지 집계하여 집계 테이블로 관리한 후 정보를 제공한다.

이 두가지의 경우중 하나는 수행속도의 저하라는 문제점을 가지고 있고 다른 하나는 Real Time 정보가 아니라는 문제점을 가지고 있다. 이러한 문제점들도 데이터의 처리시



〈그림 3〉 D-1 + D = RealTime

활용도를 높이면 쉽게 해결할 수 있다.

수학적인 집합 개념을 도입해 보면특정일자에서 전일까지의 집합에 당일의 집합을 더하면 특정일자에서 당일의 집합이 된다.이것은 〈그림 3〉로 표현될 수 있다.

예를 들어 매일 평균 수주가 2천로우씩 발생되는 수주테이블에서 사용자가 입력한 조건의 월에 대한 제품별로 수주실적 정보를 검색한다고 가정하여 보자.(단 제품수는 30개)

검색할 로우는 최소한 6만건이므로 인덱스나 클러스터를 사용하더라도 처리할 데이터의 절대량이 많기 때문에빠른 수행속도는 기대할 수 없다. 그러나 집계 테이블을 이용하여 전일까지 일별로 제품에 대한 수주실적을 집계하여 관리한다면 집계 테이블에서 검색할 로우는 900건이 되고, 당일의 수주실적은 최대한 2천로우이기 때문에 전체 검색할 로우는 2,900건이 된다.

```
SELECT PROD, SUM(QTY), SUM(AMT)
FROM(SELECT PROD, SUM(QTY) QTY, SUM(AMT) AMT FROM수주집계테이블 — 집계분
WHERE YYMMDDLIKE '1999%' —— 데이터 액세스량 900 건
GROUP BY PROD —— GROUP BY 결과30 건
UNION ALL
SELECT PROD, SUM(QTY) QTY, SUM(AMT) AMT FROM수주테이블— 당일분
WHERE YYMMDD = TO_CHAR(SYSDATE, 'YYYYMMDD' )—데이터 액세스량 2000 건
GROUP BY PROD )——GROUP BY 결과30 건
GROUP BY PROD
```

여기서 검색된 정보는 완벽한 Real Time 데이터이고 수주테이블에 클러스터를 만들면 수행속도는 더욱더 향상된다.

수행속도를 결정하는 것은 로우의 수가 아니라 데이터의 입출력 단위 수이다.

일반적으로 수행속도는 처리하고자 하는 로우의 건수에 비례하여 결정된다고 생각한다. 이것은 관계형 데이터베이스의 입출력 메커니즘을 이해하지 못하고 있기 때문이다. SQL을 실행하여 1로우를 검색하더라도 관계형 데이터베이스의 입출력 단위로 검색하여 단지 1로우만 보여줄 뿐이다.

만일 10만로우를 검색하는 경우와 1만로우를 검색하는 경우중에서 10만로우를 검색하는 것이 빠를 수 있다. 왜냐하면 1만로우의 평균 로우 길이가 50 byte이고 1만로우의 평균 로우 길이가 1천byte인 경우에 데이터의 입출력량을 비교하여 보면 100,000(50 = 5,000,000 이고 10,000(1000 = 10,000,000 이므로 실질적으로 로우가 많은 쪽이 훨씬 빠르다는 결과를 알 수 있다.

이 원리에서 이해할 수 있는 것은 특정 테이블에 대해 데이터를 처리하는 형태가 대부분 'COUNT', 'GROUP BY' 형태로 처리하는 경우는 로우의 평균 길이를 작게 가지고 갈 수 있도록 테이블을 설계하는 것이 유리하다는 것이다.

이상에서 관계형 데이터베이스의 수행속도를 향상시킬 수 있는 기본적인 사항에 대하여 설명하였다. 설명한 내용을 보면 대부분의 개발자들은 너무나 당연한 내용을 담고 있다고 생각할지 모른다. 그러나 이 내용들은 수행속도를 향상시킬 수 있는 기초이다.

모든 것이 기초가 튼튼하다면 하루 아침에 무너지지 않고 활용폭도 무한정 넓어지게 된다. 만일 자신의 시스템이 수행속도가 늦다면 앞에서 설명한 내용을 다시 한번 정독을 한 후 기본적인 사항을 지키고 있는지 점검하여 보기를 바란다.