ORACLE 9*i* 개발자 튜닝가이드 *v0.92*

(with SQLTools for Oracle)

ORACLE 9*i* 개발자 튜닝가이드 *v0.92*
Mail:heiya@nate.com
Homepage:http://myhome.naver.com/heiya
Last edited : 2003.06.10

- 목차 -

III. SQL 활용 I

IV. SQL 활용 II

V. TUNING

Ⅰ. ORACLE의 이해

1. ORACLE Optimizer

ORACLE은 Rule-Based Optimization(RBO)과 Cost-Based Optimization(CBO)를 모두 지원하고 있다. CBO의 경우,1992년 Oracle 버전 7부터 도입되었고 RBO는 간단한 규칙 위주로 최적화를 수행하는 방법으로 앞으로는 널리 사용되지 않을 것이며 ORACLE도 공식적으로 CBO 사용을 권장하고 있다.

ORACLE에서 사용자의 질의한 SQL은 다음 4단계를 거쳐서 수행된다.

> 파싱(Parser)
> 옵티마이저(Query Optimizer)
> 로우소스 생성(Row Source Generator)
> SQL 실행(SQL Execution Engine)

1) 파싱(Parser)

파싱 단계는 SQL은 구문(syntax)과 의미(semantics) 검사를 수행한다. Syntax 체크는 SQL 문장의 문법을 검증하는 단계이고, Semantic 체크는 SQL 문장내의 오브젝트가 존재하는지, 권한이 존재하는지 등을 검증한다.

이 단계가 끝나면, SQL 문은 파싱 트리(parsed tree) 형태로 변형되어 optimizer에게 넘겨진다.

2) 옵티마이저(Query Optimizer)

옵티마이저(Query Optimizer) 단계는 앞에서 넘겨받은 파싱 트리를 이용해서 최적의 실행 계획을 고른다. 이 때 SQL의 결과를 처리하기 위한 효율적인 방법을 결정하기 위해 내부적인 규칙을 사용한다.

3) 로우소스 생성(Row Source Generator)

로우소스 생성(Row Source Generator) 단계는 Optimizer에서 넘겨받은 실행 계획을 내부적으로 처리하는 자세한 방법을 생성하는 단계이다. '로우 소스'란 실행 계획을 실제로 구현하는 인터페이스 각각을 지칭하는 말로, 테이블 액세스 방법, 조인 방법, 그리고 정렬(sorting) 등을 위한 다양한 로우 소스가 제공된다. 따라서, 이 단계에서는 실행 계획에 해당하는 트리 구조의 로우 소스들이 생성된다.

4) SQL 실행(SQL Execution Engine)

SQL 실행(SQL Execution Engine) 단계는 위에서 생성된 로우 소스를 SQL 수행 엔진에서 수행해서 결과를 사용자에게 돌려주는 과정이다.

여기서 한 가지 주목할 점은, 소프트 파싱(soft parsing)과 하드 파싱(hard parsing)은 크게 옵티마이저 단계의 포함 여부에 따른 차이라는 것이다.
즉, 소프트 파싱은 이미 최적화를 한 번 수행한 SQL 질의에 대해 옵티마이저 단계와 로우 소스 생성 단계를 생략하는 것이고, 하드 파싱은 이 두 단계를 새로 수행하는 것이다. 따라서, 하드 파싱은 통계정보 접근과 실행 계획 탐색 때문에 시간이 많이 걸린다.

결국 이 차이가 주로 SQL 튜닝 전문가들이 가급적이면 하드 파싱을 피하라고 권하는 이유이다.

하드 파싱을 줄이는 방법은 V장 TUNING편에서 다루도록 하겠다.

2. SQL 튜닝과 옵티마이저의 관계

SQL 튜닝은 특정 SQL 질의의 수행 시간을 단축하기 위해 사용자가 취하는 다양한 방법을 통칭한다. SQL 튜닝의 범위는 굉장히 포괄적인데, 옵티마이저와 관련한 방법으로는 SQL 재작성, 힌트 사용, 새로운 인덱스 추가, 통계 데이터의 추가/갱신 등을 통해서 옵티마이저가 더욱더 효율적인 실행 계획을 생성하도록 하는 것이다.

1) SQL의 변경

사용자가 원하는 데이터를 질의하는 방법은 실제로 매우 다양할 수 있다. 그러나 SQL의 작성 방법에 따라 optimizer가 다른 경로로 실행 계획을 수립할 수 있다.
이 주제에 대해서는 뒤에서 다시 다루도록 하겠으나 SQL의 재작성을 통한 SQL 튜닝은 원래의 SQL 문을, 같은 결과를 내지만, 옵티마이저가 더 효과적인 실행 계획을 생성할 수 있는 SQL 문으로 바꾸는 방법이다.

2) HINT의 사용과 경로의 제어

ORACLE의 힌트 기능은 ORACLE로 하여금 사용자가 원하는 경로의 실행 계획을 만들어 내도록 하는 것이다.
이 주제 또한 뒤에 다루겠지만 SQL의 변경이나 재작성 없이 실행 경로를 제어 할 수 있는 방법이다. 그러나 CBO를 사용하는 경우에는 반드시 지정한 경로로 실행되지는 않는다. 통계

정보의 유무 등에 의해 사용자의 Hint가 무시되는 경우도 있으므로 반드시 실행 계획을 확인하도록 한다.

힌트 외에 인덱스를 사용하지 못하도록 인덱스 컬럼을 가공하는 등 실행 경로를 제한하는 방법도 있다.

그러나 이렇게 힌트를 사용하거나 인덱스 컬럼을 가공하여 작성된 SQL은 향후에도 지속적으로 영향을 미치므로 주의하여야 한다.

3) 새로운 인덱스 추가

SQL 문의 효율적인 처리를 위해서는 특정 테이블의 특정 칼럼 값을 이용해서 해당 데이터를 빨리 찾아야 하는데, 인덱스가 없기 때문에 옵티마이저가 어떤 실행 계획을 선택하더라도 그 SQL 문은 느릴 수밖에 없는 경우가 있다.

이와 같은 상황에서는 새로운 인덱스 생성을 통해서 옵티마이저가 해당 인덱스를 이용하는 새로운 실행 계획을 선택하도록 할 수 있다.

인덱스의 생성과 개수에 대한 얘기는 뒤에서 다시 다루도록 하겠다.

4) 통계 정보의 추가 및 갱신

Optimizer의 비용 산정 모듈에서는 테이블, 칼럼, 인덱스 등에 대한 통계정보를 이용해서 선택도(selectivity), 카디널리티(cardinality) 등을 구하고 이를 통해서 궁극적으로 실행 계획의 비용을 계산한다.

그런데, 만일 특정 테이블/칼럼에 대한 통계 정보가 없거나, 오래 전에 만들어진 경우는 비용 계산이 부정확하게 되고, 따라서 옵티마이저가 선택하는 실행 계획이 실제로는 안 좋은 실행 계획일 수가 있다.

이를 해결하기 위해서는 특정 통계정보를 추가하거나 새로 갱신해 주어서 옵티마이저가 정확한 비용 산정을 통해서 더 나은 실행 계획을 선택하도록 해주는 방법이다.

3. Tables에 대한 Access

ORACLE은 기본적으로 테이블의 Row에 접근하기 위해 2개의 방법을 사용한다.

1) TABLE ACCESS FULL

Full table scan은 테이블의 각 ROW를 순차적으로 읽는다. ORACLE은 full table scan의 성능을 향상시키기 위해 동시에 여러 개의 Block을 읽는다.

특히 where절이 없는 query를 수행할 때에는 반드시 full table scan을 하므로 조심하여야 한다.

2) ROWID를 통한 TABLE ACCESS

테이블에 대한 접근 성능을 향상시키기 위해서 RowID라고 하는 임의의 컬럼 값을 사용하여 각 Row에 접근 한다. RowID는 row가 저장되어 있는 물리적인 위치에 대한 정보를 담고 있다.

ORACLE은 데이터의 물리적 위치 정보를 담고 있는 RowID와 관련된 index를 사용한다. 이 인덱스를 통해 ORACLE은 RowID에 빨리 접근할 수 있으며 index에 사용된 컬럼을 찾는 query의 수행 속도를 향상 시킨다.

인덱스를 통한 Table 접근은 query의 형태에 따라 unique와 range로 나눌 수 있다.

4. SQL문의 공유

ORACLE은 SQL문을 parsing한 후 동일한 SQL문이 재실행될 때 다시 parsing하는 부하를 줄이기 위해 SQL문과 parsing된 정보를 메모리에 저장하여 활용한다. 이 정보는 System Global Area(SGA)의 일부분인 shared buffer pool내에 single shared context area에 저장되며 모든 유저가 공유한다.

이렇게 저장된 정보는 DB 내의 어느 사용자든 동일한 SQL문을 실행하게 때 다시 parsing하여 최적화된 실행경로를 생성해 내야 하는 부하를 줄일 수 있어 더 빠르게 결과값을 얻을 수 있다. 이는 성능 향상과 함께 메모리를 절약하는 방법 중 하나이다.

DBA는 효율적인 context area 사용을 위해서는 cache로 할당할 메모리의 양에 대한 적절한 값을 init.ora 내의 parameter에 지정하여야 한다. Context area는 클수록 많은 정보를 저장하고 활용할 수 있지만 시스템의 성능과 용도에 따라 반드시 그런 것은 아니므로 숙련된 엔지니어에게 의뢰하는 것이 좋다.

SQL문이 실행될 때마다 ORACLE은 먼저 context area에 동일한 문장에 대한 parsing 정보가 있는지 확인한다. 불행히도 ORACLE은 carriage return, space 그리고 대소문자를 구분하므로 이러한 문자들까지 일치하도록 하여야 다시 parsing하는 부하를 줄일 수 있다..

이러한 비교 조건을 만족시켜 shared area내의 parsing정보를 사용하기 위해서는 아래의 세가지 규칙을 따라야 한다.

1) 저장된 SQL문과 실행되는 SQL문의 모든 문자가 일치하여야 한다.

예) SELECT * FROM EMP;

앞의 예)는 다음의 문장과 전혀 다른 SQL로 인식된다.

```
ㄱ) SELECT * from EMP;
ㄴ) Select * From Emp;
ㄷ) SELECT*       FROM EMP;
```

아래 문장 또한 첫번째 문장이 두 줄로 분리되어 있어 다음 동일한 문장으로 인식되지 않는다.

```
ㄱ) Select pin from person
     where last_name = 'LAU';
ㄴ) Select pin from person where last_name = 'LAU';
```

2) 새로 실행되는 SQL문 내에서 참조하는 object가 기존 SQL문 내의 object와 동일하여야 한다.

예) 다음 예에서 각 유저는 아래의 object들을 참조한다고 하자.

USER	OBJECT명	종류
JACK	sal_limit	private synonym
	work_city	public synonym
	plant_detail	public synonym
JILL	sal_limit	private synonym
	work_city	public synonym
	plant_detail	table owner

앞의 예)를 참조하여 다음의 SQL의 공유 여부를 살펴 보자.

SQL	OBJECT Matching	이유
`select max(sal_cap) from sal_limit;`	NO	각각의 유저가 private synonym인 sal_limit을 가지고 있으므로 서로 다른 object이다.
`select count(*) from work_city where sdesc like 'NEW%';`	YES	두 USER 모두 work_city라는 public synonym을 참조하므로 동일한 SQL문이다.
`select a.sdesc, b.location from work_city a, plant_detail b where a.city_id = b.city_id;`	NO	JACK이 public synonym을 통해 JILL의 plant_detail을 참조하므로 서로 다른 object이다.
`select * from sal_limit where over_time is not null;`	NO	각각의 유저가 private synonym인 sal_limit을 가지고 있으므로 서로 다른 object이다.

3) Bind variable을 사용할 때 variable명은 모두 동일하여야 한다.

예) 다음 두 문장은 동일한 문장이다.

```
select ename, empno from emp where empno = :pEmpNo;
select ename, empno from emp where empno = :pEmpNo;
```

다음의 두 문장은 run-time 시 동일한 값이 부여된다 하더라도 다른 문장으로 인식한다.

```
select ename, empno from emp where empno = :pEmpNo;
select ename, empno from emp where empno = :v_Emp_No;
```

5. FROM절에서의 테이블 순서 (RBO만 해당)

ORACLE parser는 항상 오른쪽 테이블부터 왼쪽 테이블로 실행경로를 생성한다. 만약 from절에 여러 개의 테이블 이름이 나열된 경우 적은 수의 row를 가진 테이블을 오른쪽에 배치하여 driving 테이블로 만들어야 한다. ORACLE이 여러 개의 테이블에 대한 처리를 할 때, 내부적으로 sort나 merge procedure를 통해 테이블들이 join 된다.

먼저 첫번째 테이블을 scan하여 sort한 후 다음 테이블을 scan한다. 그 후 이 테이블에서 추출된 데이터를 첫번째 테이블에서 추출된 데이터와 함께 조합하여 결과값을 돌려준다.

예) Table TAB1 : 16,384 rows.
Table TAB2 : 1 row.

TAB2을 driving table로 하여 select 할 때.

```
SELECT COUNT(*)
  FROM TAB1, TAB2; - 0.96 seconds
```

TAB1을 driving table로 하여 select 할 때. (Poor Approach)

```
SELECT COUNT(*)
  FROM TAB2, TAB1; - 26.09 seconds
```

3개의 테이블이 조인될 때, 상위 테이블을 driving table로 선택한다. ERD 상의 상위 테이블은 그에 종속되는 많은 테이블을 가지고 있다.

예) LOCATION 테이블과 CATEGORY 테이블은 EMP 테이블의 속성 정보를 나타낸다.

Case 1	Case 2

SELECT …
  FROM LOCATION L,
  CATEGORY C,
  EMP E
WHERE E.EMP_NO BETWEEN 1000 AND 2000
  AND E.CAT_NO = C.CAT_NO
  AND E.LOCN = L.LOCN;
	SELECT …
	  FROM EMP E,
	  LOCATION L,
	  CATEGORY C
	WHERE E.EMP_NO BETWEEN 1000 AND 2000
	  AND E.CAT_NO = C.CAT_NO
	  AND E.LOCN = L.LOCN;

일반적으로 Case 1이 Case 2보다 더 효율적이다.

그러나 항상 그런 것은 아니며, 각각의 경우마다 서로 수행 성능을 보일 수 있으므로 Application에 대한 적용은 충분한 학습 후 하여야 한다.

6. Driving Table의 선택

Query 수행시 처음 읽게 되는 테이블을 Driving Table이라 하며, 이는 사용하고 있는 optimizer의 모드나 통계정보의 유무 등에 따라 결정된다.

Cost-Base Optimizer(CBO)를 사용하는 경우에는 analyze 명령어를 통해 생성된 통계정보를 이용하여 테이블의 사이즈, 인덱스의 유용성, 최소 cost를 요하는 경로 등을 선택한다.

Rule-Base Optimizer(RBO)를 사용하고 join 조건으로 사용되는 컬럼에 대해 index가 생성되어 있거나, 힌트 등에 의해 순서를 지정하지 않는 경우 **FROM**절의 오른쪽 테이블을 driving table로 결정하게 된다.

```
예) SELECT A.NAME, B.MANAGER
      FROM WORKER A,
           LODGING B
     WHERE A.LODGING = B.LODGING;
```

LODGING 컬럼에 대한 index가 사용가능 하고, WORKER 테이블에 비교 가능한 index가 없을 경우, 인덱스가 없는 WORKER 테이블이 Driving table이 된다. 이는 LODGING 테이블이 Driving table이 될 경우 WORKER 테이블을 **FULL TABLE SCAN**하게 됨으로써 발생되는 과부하를 줄이기 위한 ORACLE의 내부 메커니즘에 의한 선택이다.

7. Optimizer의 Index 선택

Cost-Based Optimizer(CBO)는 SQL문을 실행 할 때 사용 가능한 index 중 cost가 가장 적게 드는 index를 선택하여 사용한다. Index의 분포도가 좋으면 그만큼 선택도(SELECTIVITY)가 높게 된다.

예를 들어, 100row가 있는 테이블에 어떤 한 컬럼이 다른 값들과 구별이 되는 row가 80개 있을 때, 해당 컬럼에 대한 인덱스의 선택도는 80/100 = 0.80으로 높게 나타난다.

인덱스의 분포도가 낮으면 INDEX RANGE SCAN과 TABLE ACCESS BY ROWID 작업이 TABLE ACCESS FULL에 비해 많은 I/O를 발생할 수 있다.

그러나 전체적인 컬럼의 분포도가 좋더라도 특정 값에 대한 분포도가 떨어 진다면 해당 값에 대한 query는 인덱스를 사용하지 않는 것이 낫다. 일반적으로 특정 값의 분포도가 10 - 15% 이상이면 full table scan이 유리하나 row의 전체 건수, 시스템의 성능 등을 종합적으로 판단하여야 한다.

8. Oracle의 내부 Operation

ORACLE이 Query를 수행할 때 사용하는 명령어에 따라 여러 가지 내부적인 작업을 하게 된다. 아래의 표는 query를 수행할 때 수반되는 내부 작업들을 보여 준다.

명령어	ORACLE 내부 작업
ORDER BY	SORT ORDER BY
UNION	UNION-ALL
MINUS	MINUS
INTERSECT	INTERSECTION
DISTINCT, MINUS, INTERSECT, UNION	SORT UNIQUE
MIN, MAX, COUNT	SORT AGGREGATE
GROUP BY	SORT GROUP BY
ROWNUM	COUNT 또는 COUNT STOPKEY
Join된 SQL문	SORT JOIN, MERGE JOIN, NESTED LOOPS
CONNECT BY	CONNECT BY

위와 같은 ORACLE의 내부 operation을 잘 이해해야만 원치 않는 과부하를 사전에 방지하고 원하는 결과를 더 빠른 시간에 얻을 수 있다.

II. INDEX의 활용

1. INDEX를 통한 성능 향상

Index는 테이블로부터 데이터를 빠르게 조회하기 위해 사용하는 논리적이고 또한 물리적인 실체를 가진 OBJECT이다.
ORACLE은 내부적으로 B-tree(Balanced-tree) 방식의 인덱스 구조를 사용한다.

Index를 통한 데이터의 조회는 테이블 내의 데이터 양이 많을 경우 full table scan에 의한 조회 보다 빠르다. ORACLE의 optimizer는 select, update, delete시 가장 효율적인 경로를 생성하기 위해 table에 명시된 index를 사용한다. 또한 여러 테이블의 join에도 index를 사용한다.

Index를 사용함으로써 얻을 수 있는 또 다른 이점은 primary key에 의해 유일성(Uniqueness)을 보장 받을 수 있다는 것이다.
LONG이나 LONG RAW 타입의 컬럼을 제외하고는 어떤 컬럼을 이용해서든 인덱스를 만들 수 있다.

일반적으로 대용량 테이블에 만들어진 인덱스는 대단히 유용하다. 만약 작은 테이블일지라도 빈번히 join에 사용된다면 index를 활용한 성능 향상이 가능하다.

그러나 index가 일반적으로 성능의 향상을 제공하기는 하지만 이의 사용에 따른 cost가 발생한다는 것을 알아야 한다. 우선 index는 물리적인 object이므로 storage를 필요로 한다. 또한 데이터의 delete, insert, update시에 인덱스에도 동일한 작업이 필요하게 된다.
이러한 작업이 빈번히 일어나게 되면 인덱스의 효율이 나빠지기도 하므로 지속적인 관리가 필요하게 된다.

위와 같이 인덱스의 개수에 따라 수회 이상의 storage와 system에 대한 overhead가 발생하여 오히려 성능을 저하시키기도 한다.

일반적으로 테이블별로 4개 이상의 인덱스를 생성하는 것은 좋지 않다. 그러나 상황에 따라 필요한 경우 인덱스는 생성하는 것도 고려하여야 한다.

1) INDEX 사용 기준

ㄱ) 대상 컬럼 선정
-. 조건문에 자주 등장하는 컬럼
-. 분포도가 좋은 컬럼 (같은 값이 적은 컬럼)
-. 자주 JOIN에 사용되는 컬럼

ㄴ) INDEX 사용시 손해 보는 경우

-. 데이터가 적어 full table scan이 더 유리한 경우 (보통 16 block 이내인 경우)

-. 분포도가 나쁜 컬럼 (같은 값이 많은 컬럼)

-. SELECT 보다 DML 부담이 더 큰 경우

2) 결합 INDEX 사용 기준

ㄱ) 대상 컬럼 선정

-. 2개 이상의 컬럼이 자주 JOIN에 사용되는 경우

-. 인덱스만으로 결과값을 얻을 수 있는 경우

-. 자주 JOIN에 사용되는 컬럼

ㄴ) 결합 INDEX의 컬럼 순서 (나열순서와는 상관없음)

-. 사용빈도가 높은 컬럼

-. 분포도가 좋은 컬럼

-. 자주 사용되는 컬럼

2. INDEX를 이용한 작업

ORACLE은 index를 통해 접근하는데 다음과 같은 두 방식을 이용한다.

1) INDEX UNIQUE SCAN

조회하고자 하는 테이블에 인덱스가 존재하는 경우 optimizer는 query시 인덱스를 이용한다.

예) EMP 테이블에 EMP_NAME 컬럼에 EMP_PK라는 unique index와 MANAGER컬럼에 non-unique index인 EMP_IDX01이라는 두개의 인덱스가 존재한다고 하자.

```
SELECT *
  FROM EMP
 WHERE EMP_NAME = 'ROSE HILL';
```

내부적으로 위의 query를 수행하기 위해 두개의 step으로 나뉘어 진행된다.

첫번째는 먼저 EMP_PK 인덱스를 통한 **INDEX UNIQUE SCAN** 작업이 수행되어 EMP_PK에서 EMP_NAME이 'ROSE HILL'인 데이터의 물리적 위치인 RowID를 찾게 된다.

그 다음에 RowID를 이용한 **TABLE ACCESS BY ROWID** 작업이 수행되어 EMP 테이블에서 해당 row의 나머지 컬럼을 찾아 결과값을 되돌려 준다.

만약 query를 통해 요청 되는 값이 index 내의 컬럼 이라면, 첫번째 작업인 **INDEX UNIQUE SCAN**만으로도 결과값을 되돌려 줄 수 있어 더 높은 성능을 기대할 수 있다.

다음의 SQL문은 **INDEX UNIQUE SCAN** 작업만으로 수행한다.

```
SELECT EMP_NAME
  FROM EMP
 WHERE EMP_NAME = 'ROSE HILL';
```

2) INDEX RANGE SCAN

Index 컬럼에 대한 범위 또는 non-unique index의 컬럼을 이용한 query를 수행하면 **INDEX RANGE SCAN**을 통해 데이터를 조회하고 값을 되돌려 받게 된다.

```
예) SELECT EMP_NAME
      FROM EMP
     WHERE EMP_NAME LIKE 'R%';
```

위 예에서 where절 내에 EMP_NAME에 대한 범위로 조회를 하므로 unique-index인 EMP_PK를 통해 조회되더라도 **INDEX RANGE SCAN** 작업을 통해 데이터가 조회된다.

INDEX RANGE SCAN을 통해 수행되는 작업은 인덱스로부터 여러 개의 데이터를 조회하기 때문에, **INDEX UNIQUE SCAN**에 의한 것보다 비효율적이다.

조회하는 컬럼이 EMP_PK를 구성하는 EMP_NAME이므로 **INDEX RANGE SCAN** 만으로 query가 수행되고 값을 되돌려 받게 된다.

```
예) SELECT EMP_NAME
      FROM EMP
     WHERE MANAGER = 'BILL GATES';
```

위 예)와 같은 SQL은 조회하는 컬럼이 인덱스의 구성 컬럼이 아니므로 내부적으로 두 단계에 걸쳐 실행된다. 첫번째는 non-unique index인 EMP_IDX01을 **INDEX RANGE SCAN**하여 RowID를 얻게 되고, 이를 통해 테이블에 대한 **TABLE ACCESS BY ROWID**를 통해 수행되어 EMP_NAME 컬럼 값을 되돌려 준다.

여기서 non-unique index인 EMP_IDX01의 MANAGER 값이 unique할지라도 **INDEX UNIQUE SCAN**을 통해 수행되지는 않고 **INDEX RANGE SCAN**을 통해 n+1건에 대한 access가 발생한다.

그러나 아래의 경우와 같이 인덱스 컬럼에 대해 range로 조회를 하더라도 맨 앞 글자가 '%'와 같은 와일드카드 일 경우에는 non-unique index인 EMP_IDX01이 존재하더라도 인덱스를 통한 access가 일어나지 않고 **FULL TABLE SCAN**으로 조회된다.

```
예) SELECT EMP_NAME
      FROM EMP
     WHERE EMP_NAME LIKE '%GATES';
```

3. 우선 순위가 분명치 않은 INDEX의 선택

Index의 우선 순위가 분명치 않은 경우 ORACLE은 Where절에 먼저 기술된 인덱스 하나만을 이용하여 경로를 생성한다.

예) EMP 테이블의 DEPTNO 컬럼과 EMP_CAT 컬럼에 각각 non-unique index가 있다고 하자.

```
SELECT ENAME
  FROM EMP
 WHERE DEPTNO > 20
   AND EMP_CAT > 'A';
```

위 예)는 DEPTNO에 대한 인덱스만을 사용한다. Explain plan은 다음과 같다.

```
TABLE ACCESS BY ROWID ON EMP
  INDEX RANGE SCAN ON DEPT_IDX
```

4. 두개 이상의 인덱스의 MERGE

서로 다른 테이블에 EQUAL('=')로 조회되는 두개 이상의 사용 가능한 인덱스가 있는 경우 ORACLE은 run-time시 모든 인덱스를 병합(merge)하여 원하는 값을 되돌려 주기도 한다.

Unique index가 있는 경우 non-unique index 보다 상위가 된다.

위 내용은 모두 상수와 비교될 때에만 해당되며, 만약 다른 테이블의 인덱스와 비교될 경우에는 optimizer에 의해 하위로 분류되기도 한다.

만약 서로 다른 테이블에 동등한 순위의 인덱스가 있는 경우에는 optimizer가 FROM절에 쓰인 순서 등 일정한 RULE에 의해 순서를 결정한다.

만약 동등한 순위의 인덱스가 하나의 테이블 내에 존재한다면 Where절에 먼저 쓰인 컬럼의 인덱스가 우선 참조되고 아래쪽에 기술된 컬럼의 인덱스가 나중에 참조된다.

예) EMP 테이블의 DEPTNO 컬럼과 EMP_CAT 컬럼에 각각 non-unique index가 있다고 하자.

```
SELECT ENAME
  FROM EMP
 WHERE DEPTNO = 20
   AND EMP_CAT = 'A';
```

위 예)를 보면 다음과 같이 DEPTNO에 대한 인덱스를 먼저 사용하고 그 다음에 EMP_CAT에 대한 인덱스를 사용하여 각각의 결과치를 가지고 equal 비교를 하여 결과치를 돌려 준다.

```
TABLE ACCESS BY ROWID ON EMP
  AND-EQUAL
    INDEX (RANGE SCAN) ON 'DEPT_IDX' (NON-UNIQUE)
    INDEX (RANGE SCAN) ON 'CAT_IDX' (NON-UNIQUE)
```

5. 사용자에 의한 INDEX 사용 제한

둘 이상의 인덱스가 동일한 순위로 참조되어 query가 비효율적인 경로로 실행될 때, 어느 하나의 인덱스를 강제로 사용하지 못하게 함으로써 SQL문의 실행 성능을 높일 수 있다.

Character 타입의 컬럼에는 ||''을 붙이고, Number 타입의 컬럼에는 +0을 하여 좌변을 가공하게 되면 ORACLE은 해당 인덱스를 사용하지 않는 실행 계획을 세우게 된다.

```
예) SELECT ENAME
      FROM EMP
     WHERE EMPNO = 7935
       AND DEPTNO +0 = 10
       AND EMP_TYPE||'' = 'A';
```

위 SQL문처럼 인덱스의 사용을 제한하는 것은 hint의 사용처럼 현재 뿐 아니라 미래에도 영향을 미치게 된다. 즉, 현재는 EMPNO에 대한 인덱스가 다른 인덱스보다 분포도가 좋아 더 나은 결과를 보일지 몰라도 향후 데이터의 양이나 분포도가 변할 경우 dynamic하게 경로를 생성하는 CBO의 장점을 활용하지 못하게 된다. 그러므로 필요한 경우에만 잠시 사용하는 것이 좋다.

전략적으로 인덱스의 사용을 제한하는 경우를 살펴보자. 현재 EMP_TYPE 컬럼에 대해 non-unique index가 생성되어 있고, EMP_CLASS 컬럼에는 인덱스가 없다.

```
SELECT ENAME
  FROM EMP
 WHERE EMP_TYPE = 'A'
   AND EMP_CALSS = 'X';
```

Optimizer는 당연히 EMP_TYPE에 대한 인덱스를 사용한 실행계획을 수립한다. 그러나 향후에 EMP_CLASS에 대한 인덱스가 생성되면 어떻게 될 것인가?

일반적인 경우 앞에서 설명한 바와 같이 두개의 인덱스를 모두 활용한 실행계획을 생성하여 sort/merge를 통해 query를 수행한다. 이 때 한 인덱스는 분포도가 좋아 10건 미만의 결과값을 찾고, 다른 인덱스는 분포도가 떨어져 수천 건을 결과값으로 돌려 받았다면, 이를 sort/merge 하는데 걸리는 overhead는 분명히 성능 저하의 원인이 된다.

이와 같이 어느 한쪽이 항상 우수한 분포도를 보이며, 이를 확신할 수 있을 경우 아래와 같이 전략적으로 인덱스의 사용을 제한함으로써 향후 발생 가능한 overhead를 미연에 방지할 수 있다.

```
SELECT ENAME
  FROM EMP
 WHERE EMP_TYPE = 'A'
   AND EMP_CALSS||'' = 'X';
```

6. ORACLE에 의한 INDEX 사용 제한

하나의 테이블에 두개 이상의 사용 가능한 인덱스가 있고, 하나의 인덱스만 Unique이고 나머지는 non-unique 일 때 ORACLE은 unique index만을 사용한 실행계획을 생성하고 나머지 인덱스들은 완전히 무시한다.

```
예) SELECT ENAME
      FROM EMP
     WHERE EMPNO = 2362
       AND DEPTNO = 20;
```

위 SQL문의 EMP 테이블에 EMPNO에 대한 unique index와 DEPTNO에 대한 non-unique index가 있을 때, EMPNO에 대한 unique index인 EMPNO_IDX가 조회에 사용되며 두번째 조건인 DEPTNO = 20은 확인 조건으로만 사용된다. 실행 계획은 다음과 같다.

```
TABLE ACCESS BY ROWID ON EMP
    INDEX UNIQUE SCAN ON EMPNO_IDX
```

7. 인덱스 컬럼에 대한 NOT 사용 제한

Where절 내에서 인덱스 컬럼에 대해 NOT 비교문을 사용할 경우 해당 컬럼을 가공하는 것과 동일한 효과가 나타나서 ORACLE은 NOT을 만나게 되면 해당 컬럼에 대한 인덱스를 사용하지 않는 실행계획을 수립한다.

인덱스를 사용하지 못하는 경우	인덱스의 사용이 가능한 경우

SELECT …
  FROM DEPT
 WHERE DEPT_CODE != 0;
	SELECT …
	  FROM DEPT
	 WHERE DEPT_CODE > 0;

드문 경우에 ORACLE의 optimizer가 자동으로 NOT을 변환 하는 경우도 있다.

```
NOT >  to <=
NOT >= to <
NOT <  to >=
NOT <= to >
```

8. INDEX 컬럼에 대한 가공 I

Where절에서 인덱스 컬럼이 가공되어 사용될 경우 optimizer는 해당 인덱스를 사용하지 않게 되고 사용 가능한 다른 인덱스가 없다면 이 SQL문은 full-table scan을 통해 결과값을 얻게 된다.

가공된 경우	가공하지 않은 경우
SELECT … FROM DEPT WHERE SAL * 12 > 25000;	SELECT . . . FROM DEPT WHERE SAL > 25000 / 12;
SAL 컬럼에 대한 가공이 이뤄져 이 컬럼에 대한 인덱스가 있더라도 사용되지 않고 full-table scan을 하게 된다.	SAL 컬럼에 대한 인덱스가 있다면 이를 사용하여 INDEX RANGE SCAN을 하게 된다.

9. INDEX 컬럼에 대한 가공 II

ORACLE은 두개의 서로 다른 타입의 컬럼을 비교할 때 내부적으로 형변환을 하게 된다.
다음과 같이 Number형의 EMPNO 컬럼에 인덱스가 있다고 하자.

```
SELECT …
  FROM EMP
 WHERE EMPNO = '124';
```

위에서 Number와 Character가 비교 되므로 ORACLE은 내부적으로 형변환을 하여 아래와 같은 SQL로 변형되어 수행된다.

```
SELECT …
  FROM EMP
 WHERE EMPNO = TO_NUMBER('123');
```

여기서 형변환이 일어났다 하더라도 인덱스 컬럼에 대한 가공이 아니므로 인덱스를 사용하는 데에는 문제가 없다.

다음의 경우를 생각해 보자. Character형의 EMP_TYPE 컬럼에 인덱스가 있다.

```
SELECT …
  FROM EMP
 WHERE EMP_TYPE = 123;
```

위 문장은 내부 형변환에 의해 다음과 같은 문장으로 수행된다.

```
SELECT …
  FROM EMP
 WHERE TO_NUMBER(EMP_TYPE) = 123;
```

인덱스 컬럼이 가공되므로 EMP_TYPE에 대한 인덱스는 사용되지 않는다.

다음은 TYPE에 따른 비교 및 변환 관계이다.

형태	컬럼 1	컬럼 2	결과	비고
CHAR:CHAR	CHAR(10) '1234_____'	CHAR(4) '1234'	Equal	내부 형변환 없음. 컬럼 2에 space 6자리를 추가하여 10자리 비교함.
CHAR:VARCHAR2	CHAR(10) '1234_____'	VARCHAR2(10) '1234'	Not Equal	내부 형변환 없음. 4자리만 비교 후 결과값 return. COL 1 > COL 2
VARCHAR2: VARCHAR2	VARCHAR2(10) '1234'	VARCHAR2(4) '1234'	Equal	내부 형변환 없음. 4자리만 비교 후 결과값 return. COL 1 = COL 2
CHAR:상수	CHAR(10) '1234_____'	상수 '1234'	Equal	내부 형변환 없음. 컬럼 2에 space 6자리를 추가하여 10자리 비교함.
VARCHAR2:상수	VARCHAR2(10) '1234'	상수 '1234'	Equal	내부 형변환 없음. 4자리만 비교 후 결과값 return. COL 1 = COL 2
NUMBER:CHAR or VARCHAR2 or 문자열	NUMBER 1234	'1234'	Equal	CHAR, VARCHAR2 또는 문자열이 NUMBER 형으로 내부 형변환이 일어난다.
DATE: CHAR or VARCHAR2 or 문자열	DATE 01-may-03 00:00:00	'01-may-03'	Equal	CHAR, VARCHAR2 또는 문자열이 00:00:00 붙은 DATE 형으로 내부 형변환이 일어난다.

10. 결합 INDEX의 선행 컬럼 사용과 SKIP SCAN의 활용

ORACLE 9i 이전 버전에서는 여러 컬럼으로 구성된 결합 인덱스가 있을 때, 컬럼 구성상 맨 앞에 있는 컬럼이 Where절에 사용되지 않으면 해당 인덱스를 이용한 실행 계획은 생성되지 않았다.

결합 인덱스를 사용하기 위해서는 where절에서 선행 컬럼이 가공되지 않은 채 사용되어야 하며 equal 비교로 사용되는 것이 성능을 향상 시키는 한 방법이었다.
처음 컬럼이 equal비교가 되지 않으면 그 뒤에 오는 컬럼은 사용하지 않거나 range scan이 되어 효율이 떨어지게 된다.

그러나 9i부터는 이러한 고민이 해결되었다. 불필요하게 full table scan에 의한 병목 현상을 Oracle9i Database의 **INDEX SKIP SCAN** 기능을 사용하면 해결 할 수 있다.

다음과 같이 EMP 테이블을 한번 생각해 보자.

```
CREATE TABLE SCOTT.EMP
(
  EMPNO          NUMBER(4) NOT NULL,
  ENAME          VARCHAR2(10),
  JOB            VARCHAR2(9),
  MGR            NUMBER(4),
  HIREDATE       DATE,
  SAL            NUMBER(7,2),
  COMM           NUMBER(7,2),
  DEPTNO         NUMBER(2)
);
```

그리고, 이 테이블이 다음과 같은 인덱스를 갖고 있다고 하자.

```
CREATE INDEX SCOTT.EMP_IDX01 ON SCOTT.EMP (DEPTNO, JOB, ENAME);
```

위의 인덱스는 DEPT 컬럼에 대해 query를 하는 경우에 참조하게 됩니다. 그러나, 다음과 같은 query에서는 사용되지 않는다.

```
SELECT PATIENT_SSN, GROUP_NUMBER
  FROM SCOTT.EMP
 WHERE ENAME = 'SMITH';
```

Oracle 9i 이전 버전에서는 사용 가능한 인덱스 EMP_IDX01의 선행 컬럼이 사용되지 않기 때문에 full table scan을 통해 결과값을 되돌려 준다.

Oracle 9i의 optimizer가 INDEX SKIP SCAN을 사용하기로 결정하면, 결합인덱스의 선행 컬럼인 DEPTNO의 값 들에 대해 샘플링을 하게 되고 각각의 DEPTNO에 대해 DEPTNO가 추가된 SQL을 내부

적으로 실행 하게 된다.

INDEX SKIP SCAN은 Cost-Base Optimizer(CBO)를 사용해야만 한다. 만약 Rule-Base Optimzer (RBO)가 사용되고 있다면 INDEX SKIP SCAN은 사용할 수 없다.

8i와 9i에서 확인한 결과를 다음과 같이 표로 정리하였다.

표) INDEX SKIP SCAN의 결과

SQL	INDEX의 사용		비고
	8i	9i	
SELECT * FROM SCOTT.EMP WHERE ENAME = 'SMITH';	No	Yes	결합인덱스 EMP_IDX01의 선행 컬럼 DEPTNO가 아닌 세번째 컬럼 ENAME을 사용함.
SELECT DEPTNO FROM SCOTT.EMP WHERE JOB = 'SALESMAN' AND ENAME = 'ALEN';	No	Yes	결합인덱스 EMP_IDX01의 선행 컬럼 DEPTNO가 아닌 두번째, 세번째 컬럼 JOB, ENAME을 사용함
SELECT MAX(ENAME) FROM SCOTT.EMP WHERE JOB = 'SALESMAN';	No	Yes	결합인덱스 EMP_IDX01의 선행 컬럼 DEPTNO가 아닌 두번째 컬럼 JOB을 사용함.
SELECT **/*+ RULE */** MAX(ENAME) FROM SCOTT.EMP WHERE JOB = 'SALESMAN';	No	No	RULE 힌트를 사용하여 CBO가 아닌 RBO로 작동하므로 INDEX SKIP SCAN을 사용하지 못한다.
SELECT * FROM SCOTT.EMP WHERE DEPTNO = 20	Yes	Yes	결합인덱스 EMP_IDX01의 선행 컬럼인 DEPTNO를 사용함.
SELECT * FROM SCOTT.EMP WHERE DEPTNO > 0 AND ENAME = 'CLARK';	Yes	Yes	결합인덱스 EMP_IDX01의 선행 컬럼인 DEPTNO를 사용할 수 있게 조건을 줌.

11. 인덱스 컬럼에 대한 IS NULL / IS NOT NULL의 사용 제한

인덱스로 사용되는 컬럼이 NULL값을 가질 경우 해당 row에 대해서는 인덱스를 구성하지 않으므로 IS NULL 또는 IS NOT NULL을 통한 비교가 불가능하다.

그러므로 optimizer는 해당 인덱스를 사용하지 않는 실행 경로를 생성하게 된다. 그러나 결합 인덱스의 경우 하나의 컬럼이라도 NULL이 아니면 인덱스에 포함된다.

만약 어느 테이블의 컬럼 COL1과 COL2에 unique index가 생성되어 있고, 값이 (123, null) 인 row가 이미 존재할 때 (123, null)값을 가진 row는 unique에 위배되어 입력되지 않는다.

그러나 값이 (null, null)인 row는 unique index에 저장되지 않고 제외되므로 수 천 건이라도 입력이 가능하다.

예) DEPT_CODE에 대한 인덱스가 있는 테이블 DEPARTMENT가 있다.

Case 1	Case 2
SELECT ... FROM DEPARTMENT WHERE DEPT_CODE **IS NOT NULL**;	SELECT ... FROM DEPARTMENT WHERE DEPT_CODE >= 0;

Case 1에서는 DEPT_CODE에 대한 인덱스를 참조하지 않고, Case 2의 경우는 DEPT_CODE에 대한 인덱스를 RANGE SCAN하게 된다.

12. 인덱스가 있는 경우 UNION의 사용

각각의 인덱스가 있는 컬럼에 대한 OR 검색은 때때로 optimizer로 하여금 full table scan을 하게 하는 경우가 있다.

다음의 예를 살펴보자.

Case 1	Case 2
SELECT LOC_ID, LOC_DESC, REGION FROM LOCATION WHERE LOC_ID = 10 OR REGION = 'MELBOURNE';	SELECT LOC_ID, LOC_DESC, REGION FROM LOCATION WHERE LOC_ID = 10 UNION SELECT LOC_ID, LOC_DESC, REGION FROM LOCATION WHERE REGION = 'MELBOURNE';
SELECT STATEMENT Optimizer=CHOOSE TABLE ACCESS (FULL) OF 'LOCATION'	SELECT STATEMENT Optimizer=CHOOSE SORT (UNIQUE) UNION-ALL TABLE ACCESS (BY INDEX ROWID) OF 'LOCATION' INDEX (RANGE SCAN) OF 'LOCATION_LOCID' TABLE ACCESS (BY INDEX ROWID) OF 'LOCATION' INDEX (RANGE SCAN) OF 'LOCATION_REGION'

여기서 반드시 확인할 사항이 있다. ORACLE optimizer는 버전에 따라 Case 1의 실행 계획이 아래와 같이 두개의 인덱스를 모두 사용하도록 풀릴 수도 있으므로 반드시 실행 계획을 참조하여 예상치 못한 성능 저하를 막아야 한다.

```
SELECT STATEMENT Optimizer=CHOOSE
  CONCATENATION
    TABLE ACCESS (BY INDEX ROWID) OF 'LOCATION'
      INDEX (RANGE SCAN) OF 'LOCATION_LOCID'
    TABLE ACCESS (BY INDEX ROWID) OF 'LOCATION'
      INDEX (RANGE SCAN) OF 'LOCATION_REGION'
```

III. SQL 활용 I

1. WHERE절 내에서의 JOIN의 위치

테이블 조인은 WHERE절의 조건 보다 먼저 기술되는 것이 좋다. 이는 SQL Parser에 의해 SQL이 해석될 때 WHERE절의 조건이 밑에서부터 위로 해석이 되기 때문이다.

그러므로 아래 부분에서 건수를 줄여 주면 위쪽에서 처리하는 건수가 적어지므로 더 효율적이 된다. 그러나 이는 ORACLE 버전에 따라 OPTIMIZER가 서로 다른 실행 경로를 생성하고, 데이터의 건수나 통계의 유무등에 의해 생성되는 경로가 달라지므로 반드시 실행계획을 본 후에 실행 시키는 것이 좋다.

예) * 비효율적인 경우 (Total CPU = 156.3 Sec)

```
SELECT …
  FROM EMP E
 WHERE SAL > 50000
   AND JOB = 'MANAGER'
   AND 25 < (SELECT COUNT(*)
               FROM EMP
              WHERE MGR = E.EMPNO);
```

* 효율적인 경우 (Total CPU = 10.6 Sec)

```
SELECT …
  FROM EMP E
 WHERE 25 < (SELECT COUNT(*)
               FROM EMP
              WHERE MGR = E.EMPNO )
   AND SAL > 50000
   AND JOB = 'MANAGER';
```

2. EXISTS 대신 JOIN의 사용

일반적으로 sub-query 보다는 다음과 같이 join을 하는 것이 더 좋다.:

```
SELECT …
  FROM EMP E
 WHERE EXISTS (SELECT 'X'
                 FROM DEPT D
                WHERE D.DEPT_NO = E.DEPT_NO
                AND D.DEPT_CAT = 'A');
```

아래와 같이 하면 성능 향상에 도움이 된다.:

```
SELECT ENAME
  FROM DEPT D, EMP E
 WHERE E.DEPT_NO = D.DEPT_NO
   AND D.DEPT_CAT = 'A';
```

3. 관계가 없는 테이블들에 대한 단순 결합

Relation이 없는 여러 개의 테이블에서 단순한 조회를 하는 경우 단순 결합을 통해 한번에 처리함으로써 효율을 증대 시킬 수 있다.

```
예) SELECT NAME
      FROM EMP
     WHERE EMP_NO = 1234;

    SELECT NAME
      FROM DEPT
     WHERE DEPT_NO = 10;

    SELECT NAME
      FROM CAT
     WHERE CAT_TYPE = 'RD';
```

위의 3가지 SQL문을 DUAL이라는 DUMMY 테이블을 이용하여 아래와 같이 단순 결합을 함으로써 효율성을 증대 시킬 수 있다. :

```
SELECT E.NAME, D.NAME, C.NAME
  FROM CAT  C, DEPT  D, EMP  E, DUAL X
 WHERE NVL('X', X.DUMMY) = NVL('X', E.ROWID (+))
   AND NVL('X', X.DUMMY) = NVL('X', D.ROWID (+))
   AND NVL('X', X.DUMMY) = NVL('X', C.ROWID (+))
   AND E.EMP_NO (+) = 1234
   AND D.DEPT_NO (+) = 10
   AND C.CAT_TYPE (+) = 'RD';
```

4. Equal 비교와 범위 비교

한 테이블에 대해 equal 비교와 범위 비교를 동시에 할 경우 ORACLE은 이 인덱스 들에 대해 merge를 하지 않는다.

예) EMP 테이블의 DEPTNO 컬럼과 EMP_CAT 컬럼에 각각 non-unique index가 있다고 하자.

```
SELECT ENAME
  FROM EMP
 WHERE DEPTNO > 20
   AND EMP_CAT = 'A';
```

위 query는 EMP_CAT에 대한 인덱스만을 사용한다. Explain plan은 다음과 같다.

```
TABLE ACCESS BY ROWID ON EMP
  INDEX RANGE SCAN ON CAT_IDX
```

5. 비교문 사용하기 (> 와 >=)

정수형 컬럼 DEPTNO에 대해 인덱스가 있을 때 아래의 문장을 비교해 보자.

Case 1	Case 2
SELECT *	SELECT *
FROM EMP	FROM EMP
WHERE DEPTNO > 3	WHERE DEPTNO >= 4

Case 1에서는 조건이 DEPTNO > 3 이므로 인덱스에서 DEPTNO가 3인 row부터 scan을 시작하고, Case 2에서는 DEPTNO가 4인 row부터 scan을 하게 된다. 만약 DEPTNO가 3인 row가 많다면 scan시 그만큼의 I/O가 추가로 발생하게 되므로 결과값 추출에 더 오랜 시간이 걸리게 된다.

6. IN 대신 EXISTS의 활용

ERD 상에서 base가 되는 테이블에 대한 query는 select할 때 여러 테이블과 join을 하는 경우가 많다. 이러한 경우 IN과 Sub-query를 사용하는 것 보다 EXISTS나 NOT EXISTS를 사용하는 것이 더 나은 성능을 보여주는 경우가 많다.

예) * 비효율적인 경우

```
SELECT *
  FROM EMP  (Base Table)
 WHERE EMPNO > 0
   AND DEPTNO IN (SELECT DEPTNO
                    FROM DEPT
                   WHERE LOC = 'MELB');
```

* 효율적인 경우

```
SELECT *
  FROM EMP E
 WHERE EMPNO > 0
   AND EXISTS (SELECT 'x'
                 FROM DEPT D
                WHERE D.DEPTNO = E.DEPTNO
                  AND D.LOC = 'MELB');
```

7. NOT IN 대신 NOT EXISTS의 활용

아래와 같이 Sub-query문에서 NOT IN은 내부적으로 sort와 merge를 수반한다.

NOT IN을 사용하면 대체적으로 가장 효율이 나쁜데, 이는 sub-query select에 대상이 되는 테이블을 강제로 full table scan 하도록 하기 때문이다. NOT IN 보다는 Outer Join 이나 NOT

EXISTS를 사용하는 것이 좋다.

```
SELECT …
  FROM EMP
 WHERE DEPT_NO NOT IN (SELECT DEPT_NO
                         FROM DEPT
                        WHERE DEPT_CAT = 'A');
```

위 문장의 성능을 향상 시키기 위해서는 아래와 같이 변경하여야 한다.:

Case 1	Case 2

SELECT …
  FROM EMP A, DEPT B
 WHERE A.DEPT_NO = B.DEPT_NO (+)
  AND B.DEPT_NO IS NULL
  AND B.DEPT_CAT(+) = 'A';
	SELECT …
	  FROM EMP E
	 WHERE NOT EXISTS (SELECT 'X'
	  FROM DEPT D
	  WHERE D.DEPT_NO = E.DEPT_NO
	  AND D.DEPT_CAT = 'A');

위에서 Case 1보다는 Case 2가 조금 더 나은 결과를 보인다.

8. DISTINCT 대신 EXISTS의 활용

유일성을 확보하기 위한 DISTINCT의 사용을 방지하기 위해서 아래와 같이 1:M 관계에서의 select에서는 EXISTS를 사용해야 한다.

예) * 비효율적인 방법

```
SELECT DISTINCT DEPT_NO, DEPT_NAME
  FROM DEPT D, EMP E
 WHERE D.DEPT_NO = E.DEPT_NO;
```

* 효율적인 방법

```
SELECT DEPT_NO, DEPT_NAME
  FROM DEPT D
 WHERE EXISTS (SELECT 'X'
                 FROM EMP E
                WHERE E.DEPT_NO = D.DEPT_NO);
```

EXISTS가 더 빠르게 결과값을 가져 올 수 있는 이유는 RBDMS Kernel이 sub-query 내에서 만족시키는 값을 하나 찾게 되면 바로 그 sub-query를 종료 시켜 다음 query가 진행되도록 하기 때문이다.

9. 중복되는 레코드의 삭제 방법

중복되는 레코드를 효율적으로 삭제하는 방법은 아래와 같이 RowID를 활용한 방법이다.

이는 개발 과정에서 발생하는 데이터의 중복을 배제하고 Primary키 등 제약 조건을 걸고자 할 때 활용할 수 있다.

```
예) DELETE FROM EMP E
    WHERE E.ROWID > (SELECT MIN(X.ROWID)
                     FROM EMP X
                     WHERE X.EMP_NO = E.EMP_NO);
```

위 예에서 MIN()의 사용은 경우에 따라 MAX()로도 사용할 수 있다. 그러나 중복되는 데이터 중 지워야 하는 것을 지정하고자 할 때에는 그에 맞게 SQL문을 수정하여야 함을 잊지 말자.

10. Row수 COUNT 하기

일반적인 믿음과 달리 COUNT(*)가 COUNT(1)보다 빠르다.

만약 인덱스를 통해 COUNT 한 값을 추출하고자 할 때에는 인덱스로 잡혀있는 컬럼을 COUNT(EMP)와 같이 추출하는 것이 가장 빠르게 결과값을 얻을 수 있다.

11. Table Alias의 사용

여러 개의 테이블에 대한 Query시 항상 테이블에 대한 alias를 사용하고, 각각의 컬럼에 alias를 붙여 사용하는 것이 좋다.

ORACLE이 dictionary에서 해당 컬럼이 어느 테이블에 있는지를 찾지 않아도 되므로 parsing 시간을 줄일 수 있고, 컬럼에 대한 혼동을 미연에 방지 할 수 있다.

12. WHERE와 HAVING의 차이

자주 사용하지는 않지만 간혹 HAVING을 WHERE 대신 사용하는 경우가 있다. 그러나 SELECT문에서 HAVING을 WHERE 대신 사용하는 것은 피하는 것이 좋다.

HAVING은 fetch된 row들에 대한 filter 역할을 한다. 여기에는 sort나 sum 등의 작업이 수반된다. 만약 select 하고자 하는 데이터를 일정 조건에 따라 추출하고자 할 경우에는 where절을 사용하여 HAVING을 사용함으로써 발생할 수 있는 overhead를 줄여주는 것이 좋다.

예) * 비효율적인 경우

```
SELECT REGION, AVG(LOC_SIZE)
  FROM LOCATION
 GROUP BY REGION
HAVING REGION != 'SYDNEY'
   AND REGION != 'PERTH';
```

* 효율적인 경우

```
SELECT REGION, AVG(LOC_SIZE)
  FROM LOCATION
 WHERE REGION != 'SYDNEY'
   AND REGION != 'PERTH'
 GROUP BY REGION;
```

13. SELECT절에서 Asterisk('*') 사용

Dynamic SQL 컬럼 '*'는 테이블의 모든 컬럼을 참조할 수 있게 해 준다. 그러나 이러한 '*'는 값을 되돌려 줄 때 테이블의 모든 컬럼을 변환하여 반환하므로 매우 비효율적이다. SQL Parser는 Data Dictionary에서 해당 테이블에 대한 모든 컬럼의 이름을 읽어서 SQL 명령문 내의 '*'을 대체하는 작업을 한다.

비록 0.01초 밖에 더 걸리지 않는 작업일 지라도 여러 번 반복하면 많은 시간이 걸릴 수도 있으므로 되도록 Asterisk(*)를 사용하지 않는 것이 좋다.

14. UNION-ALL의 활용

두개의 query에 대해서 UNION을 사용할 때, 각각의 query에 의한 결과값이 UNION-ALL에 의해 합쳐지고 다시 내부 작업인 SORT UNIQUE 작업에 의해 최종 결과값을 사용자에게 되돌려 준다.

이 때 UNION 대신 UNION-ALL을 사용하게 되면 SORT UNIQUE 작업은 불필요하게 되며, 그만큼의 시간을 줄일 수 있고 수행 성능을 향상시킬 수 있다.
이는 SORT가 필요하지 않은 경우에만 가능하므로 정확히 확인하고 사용하여야 한다.

Case 1	Case 2

SELECT ACCT_NUM, BALANCE_AMT
  FROM DEBIT_TRANSACTIONS
 WHERE TRAN_DATE = '31-DEC-95'
UNION
SELECT ACCT_NUM, BALANCE_AMT
  FROM CREDIT_TRANSACTIONS
 WHERE TRAN_DATE = '31-DEC-95'
	SELECT ACCT_NUM, BALANCE_AMT
	  FROM DEBIT_TRANSACTIONS
	 WHERE TRAN_DATE = '31-DEC-95'
	UNION ALL
	SELECT ACCT_NUM, BALANCE_AMT
	  FROM CREDIT_TRANSACTIONS
	 WHERE TRAN_DATE = '31-DEC-95';

Case 1의 경우에서 결과값이 서로 다르다는 것을 알고 있다면 Case 2의 경우처럼 UNION-ALL을 사용하는 것이 좋다.

15. ORDER BY를 사용하지 않고 정렬하기

ORDER BY을 사용할 때 인덱스를 사용하여 sort를 하지 않고 정렬된 결과값을 얻고자 할 때에는 다음의 두 조건을 만족하여야 한다.

>> ORDER BY에 사용된 모든 컬럼이 동일한 순서로 하나의 인덱스로 만들어져 있어야 한다.

>> ORDER BY에 사용된 모든 컬럼은 테이블 정의에 반드시 NOT NULL이어야 한다. Null값은 인덱스에 저장되지 않는다는 것을 기억하기 바란다.

Where절의 인덱스와 ORDER BY절의 인덱스는 동시에 사용될 수 없다.

예) 다음의 컬럼을 갖는 DEPT 테이블이 있다.

```
DEPT_CODE PK NOT NULL
DEPT_DESC    NOT NULL
DEPT_TYPE    NULL

NON UNIQUE INDEX DEPT_IDX ON DEPT (DEPT_TYPE)
```

Case 1	Case 2
SELECT DEPT_CODE FROM DEPT **ORDER BY** DEPT_TYPE;	SELECT DEPT_CODE FROM DEPT **WHERE** DEPT_TYPE > 0;
SORT ORDER BY TABLE ACCESS FULL	TABLE ACCESS BY ROWID ON DEPT INDEX RANGE SCAN ON DEPT_IDX

Case 2에서처럼 의미 없는 Where에 의해 ORDER BY와 같이 sort 과정을 거치지 않고도 동일한 결과를 얻을 수 있다. 'WHERE DEPT_TYPE > 0' 절에 의해 optimizer는 인덱스를 활용한 INDEX RANGE SCAN을 하게 되어 결과적으로 DEPT_TYPE의 순으로 정렬된 결과값을 얻게 된다.

16. WHERE절에서 주의할 사항

아래처럼 몇몇 경우 Where절에 사용하는 비교문에 의해 인덱스를 사용할 수 없는 경우가 있다.

No.	인덱스를 쓰지 못하는 경우	인덱스를 쓰기 위해 변경된 경우
1	SELECT ACCOUNT_NAME FROM TRANSACTION WHERE AMOUNT != 0;	SELECT ACCOUNT_NAME FROM TRANSACTION WHERE AMOUNT > 0 UNION ALL SELECT ACCOUNT_NAME FROM TRANSACTION WHERE AMOUNT < 0;
2	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE ACCOUNT_NAME||ACCOUNT_TYPE = 'AMEXA' ;	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE ACCOUNT_NAME = 'AMEX' AND ACCOUNT_TYPE = 'A;
3	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE AMOUNT + 3000 < 5000;	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE AMOUNT < 2000;
4	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE **ACCOUNT_NAME** = NVL(:ACC_NAME, **ACCOUNT_NAME**);	SELECT ACCOUNT_NAME, AMOUNT FROM TRANSACTION WHERE ACCOUNT_NAME LIKE NVL(:ACC_NAME, '%');

마지막 예처럼 인덱스가 걸린 동일한 컬럼 간의 비교에서는 인덱스를 사용할 수 없어 full-table scan을 유발한다.

17. IN의 활용

IN을 이용하여 값을 나열하게 되면 optimizer는 비교치 각각에 대해 scan을 하고, 결과치를 concatenation하여 최종 결과를 사용자에게 돌려 준다.

```
SELECT *
  FROM EMP
 WHERE MANAGER IN ( 'BILL GATES' ,  'KEN MULLER' );
```

Optimizer는 위 SQL문을 아래와 같이 해석하여 실행한다.

```
SELECT *
  FROM EMP
 WHERE MANAGER =  'BILL GATES'
    OR MANAGER =  'KEN MULLER' ;
```

위 query를 해석할 때 optimizer는 각각의 비교치에 대해 MANAGER 컬럼에 대한 인덱스를

INDEX RANGE SCAN하게 된다. Index scan을 통해 추출된 RowID를 가지고 테이블을 access하여 각각의 결과값을 추출하고, 이 값들을 다시 CONCATENATION하여 사용자에게 돌려주게 된다.

Explain Plan을 살펴보면 아래와 같다.

```
SELECT STATEMENT Optimizer=CHOOSE
  CONCATENATION
    TABLE ACCESS (BY INDEX ROWID) OF EMP
      INDEX (RANGE SCAN) OF EMP_IDX01 (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF EMP
      INDEX (RANGE SCAN) OF EMP_IDX01 (NON-UNIQUE)
```

18. DATE 사용시 주의점

DATE형의 컬럼에 대한 작업시 소수점 5자리 이하의 숫자에 대해서는 작업을 하지 않도록 한다. 소수점 6자리 숫자를 더하게 되면 다음 날짜에 대한 값을 결과로 받게 된다.

SQL문	결과값
SELECT TO_DATE('01-MAY-93') + 0.99999 FROM DUAL;	'01-MAY-03 23:59:59'
SELECT TO_DATE('01-MAY-93') + 0.999999 FROM DUAL;	'02-MAY-03 00:00:00'

IV. SQL 활용 II

1. GROUP BY의 대상 건수 줄이기

GROUP BY를 사용하는 query는 대상 건수를 줄여줌으로써 성능을 향상 시킬 수 있다.
아래와 같은 경우를 생각해 보자.

Case 1	Case 2
SELECT JOB, AVG(SAL) FROM EMP GROUP BY JOB HAVING JOB = 'PRESIDENT' OR JOB = 'MANAGER' ;	SELECT JOB, AVG(SAL) FROM EMP WHERE JOB = 'PRESIDENT' OR JOB = 'MANAGER' GROUP BY JOB;
SELECT STATEMENT Optimizer=CHOOSE FILTER SORT (GROUP BY) TABLE ACCESS (FULL) OF 'EMP'	SELECT STATEMENT Optimizer=CHOOSE SORT (GROUP BY) CONCATENATION INDEX (RANGE SCAN) OF 'EMP_IDX' (NON-UNIQUE) INDEX (RANGE SCAN) OF 'EMP_IDX' (NON-UNIQUE)

Case 1에서는 테이블의 모든 데이터에 대해 sort한 뒤 그 결과 값 중 JOB이 PRESIDENT와 MANAGER인 사람을 골라 평균 급여를 구했다.
그러나 Case 2에서는 우선 JOB이 PRESIDENT와 MANAGER인 사람을 먼저 가져온 다음 그들의 평균 급여를 구했다. 결국 데이터가 많아지면 많아 질수록 성능 차이는 많아지게 된다.

위에서와 같이 극단적인 경우가 없다고는 할 수 없다. 항상 실행 계획을 확인하는 습관을 들이는 것이 좋다.

2. 불필요한 작업을 유발하는 명령어의 사용 제한

DISTINCT, UNION, MINUS, INTERESECT, ORDER BY, GROUP BY 등의 명령어를 사용한 SQL문은 ORACLE 엔진에 부하를 주고 다량의 resource를 사용하는 sort 작업을 유발한다. 일반적으로 DISTINCT는 한번의 sort 작업을 거치고, 나머지 연산자를 처리하기 위해서는 최소한 2회 이상의 작업을 수행한다.

앞에서 살펴 본 ORACLE의 내부 Operation에서와 같이 두개의 query를 UNION으로 처리할 때 각각의 query에 대해 결과값을 추출한 다음 UNION-ALL을 통한 병합 과정을 거치고, 이 결과를 다시 SORT UNIQUE를 통해 최종 결과값을 추출하여 사용자에게 돌려준다.
Sort의 회수가 많을수록 query의 수행으로 인한 시스템 부하는 커지게 된다. 집합 연산을 하는 대부분의 query는 다른 방법으로 처리가 가능하므로 되도록 UNION, MINUS, INTERSECT와 같은 집합 연산자의 사용을 자제하여야 한다.

3. DECODE 함수를 활용한 내부 처리 단축

DECODE 함수를 활용하여 동일한 row에 대해 다시 읽는 것을 방지하거나 동일한 테이블에 대한 join을 피할 수 있다.

```
예) SELECT COUNT(*), SUM(SAL)
      FROM EMP
     WHERE DEPT_NO = 0020
       AND ENAME LIKE 'SMITH%';

    SELECT COUNT(*), SUM(SAL)
      FROM EMP
     WHERE DEPT_NO = 0030
       AND ENAME LIKE 'SMITH%';
```

DECODE 함수를 활용하여 위의 결과와 동일한 결과를 한번에 가져 올 수 있다.:

```
   SELECT COUNT(DECODE(DEPT_NO, 0020, 1, NULL))   D20_CNT,
          SUM  (DECODE(DEPT_NO, 0020, SAL, NULL)) D20_SAL,
          COUNT(DECODE(DEPT_NO, 0030, 1, NULL))   D30_CNT,
          SUM  (DECODE(DEPT_NO, 0030, SAL, NULL)) D30_SAL
     FROM EMP
    WHERE ENAME LIKE 'SMITH%';
```

이와 같이 DECODE는 GROUP BY 나 ORDER BY 절에서도 사용할 수 있다.

4. DELETE 대신 TRUNCATE의 사용

일반적인 경우 테이블 내의 데이터를 삭제하고자 할 때 ORACLE은 Rollback 세그먼트에 기존 데이터의 정보(Before image)가 저장한다. Transaction을 Commit 명령어를 실행하여 종료하지 않으면 ORACLE은 이 정보를 가지고 원래의 상태로 데이터를 Restore하게 된다.

TRUNCATE 명령어를 사용하면 undo 정보를 생성하지 않는다. DELETE보다 빠르고 Resource를 덜 사용하는 장점이 있지만 한번 truncate되면 데이터를 복구할 방법이 없으므로 주의하는 것도 잊지 말아야 한다.

5. Database에 대한 transaction 회수 줄이기

SQL문이 실행될 때마다 ORACLE은 parsing, 인덱스 확인, 변수값 할당, 데이터 읽기 등 매우 많은 내부 process를 수행한다. 그러므로 Database에 대한 접근을 적게 할수록 부하는 감소하고 효율성은 올라 간다.

예) 아래에 사원번호가 0342, 0291인 두 사원에 대한 정보를 가져오는 3가지 서로 다른 방법이 있다.

1) 두번에 걸쳐 데이터를 조회하는 방법

```
SELECT EMP_NAME, SALARY, GRADE
  FROM EMP
 WHERE EMP_NO = 0342;

SELECT EMP_NAME, SALARY, GRADE
  FROM EMP
 WHERE EMP_NO = 0291;
```

2) Cursor를 공유하는 방법 (Next Most Efficient)

```
DECLARE CURSOR C1( E_NO NUMBER)
IS
  SELECT EMP_NAME, SALARY, GRADE
    FROM EMP
   WHERE EMP_NO = E_NO;
BEGIN
  OPEN C1(342);
  FETCH  C1 INTO …,…,…;

  OPEN C1(291);
  FETCH  C1 INTO …,…,…;

  CLOSE C1;
END;
```

3) 하나의 SQL로 두가지 정보를 조회하는 방법 (Most Efficient)

```
SELECT A.EMP_NAME, A.SALARY, A.GRADE,
       B.EMP_NAME, B.SALARY, B.GRADE
  FROM EMP A, EMP B
 WHERE A.EMP_NO = 0342
   AND B.EMP_NO = 0291;
```

참고 - 한번에 가져올 수 있는 row의 수를 증가시켜 물리적인 호출 회수를 감소시키려면 SQL*Plus, SQL*Forms 그리고 Pro*C에서 ARRAYSIZE 파라미터의 값을 증가 시키면 된다. 권장값은 200.

6. COMMIT 명령어의 실행

가능하면 COMMIT 명령어를 자주 실행해 주는 것이 좋다. COMMIT 명령어를 자주 실행하는 것은 프로그램의 성능을 향상시키고 아래의 자원들을 ORACLE에 반환함으로써 Resource의 필요량이 최소화 되기 때문이다.:

> Transaction을 UNDO하기 위해 Rollback segment에 기록된 정보
> 명령어가 수행되는 도중에 걸린 LOCK
> Redo log buffer cache
> 위 세가지 Resource를 관리하기 위한 ORACLE 메커니즘에 따른 부하

그러나 여기서 한가지 주의할 점이 있다. 만일 사용중인 테이블의 데이터에 대해 UPDATE, DELETE 등을 수행하며 COMMIT을 자주 수행할 경우, 사용자에게 아래와 같은 에러 메시지가 보일 수 있다.

ORA-01555 snapshot too old: rollback segment number string with name "string" too small

이 오류는 Rollback segment와는 무관하게 작업자가 수행한 update, delete 등의 명령어에 의해 변경된 데이터의 before image를 참조하던 사용자가 commit에 의해 없어진 정보를 참조하고자 할 때 나타난다.
그러므로 운영중인 테이블에 대한 작업은 주의를 요한다.

7. Query시 테이블에 대한 참조

Query시 테이블에 대한 access 회수를 최소화 함으로써 성능을 향상 시킬 수 있다. 특히 Sub-query를 포함하거나 여러 컬럼에 대한 update를 수행할 때 작업 시간을 단축할 수 있다.

1) Sub-query

ㄱ) 비효율적인 경우

```
SELECT TAB_NAME
  FROM TABLES
 WHERE TAB_NAME = (SELECT TAB_NAME
                     FROM TAB_COLUMNS
                    WHERE VERSION = 604)
   AND DB_VER = (SELECT DB_VER
                   FROM TAB_COLUMNS
                  WHERE VERSION = 604);
```

ㄴ) 효율적인 경우

```
SELECT TAB_NAME
  FROM TABLES
 WHERE (TAB_NAME, DB_VER) = (SELECT TAB_NAME, DB_VER
                               FROM TAB_COLUMNS
                              WHERE VERSION = 604);
```

2) Multi-column UPDATE

ㄱ) 비효율적인 경우

```
UPDATE EMP
   SET EMP_CAT = (SELECT MAX(CATEGORY)
                    FROM EMP_CATEGORIES),
       SAL_RANGE = (SELECT MAX(SAL_RANGE)
                      FROM EMP_CATEGORIES )
 WHERE EMP_DEPT  = 0020;
```

ㄴ) 효율적인 경우

```
UPDATE EMP
   SET (EMP_CAT, SAL_RANGE) = (SELECT MAX(CATEGORY), MAX(SAL_RANGE)
                                 FROM EMP_CATEGORIES)
 WHERE EMP_DEPT = 0020;
```

8. Explicit Cursor의 사용

SELECT문에 사용하는 implicit cursor는 두 번의 call을 데이터베이스에 하게 된다. 첫번째는 데이터를 fetch하기 위한 call이고, 그 다음에 TOO MANY ROWS 오류를 check하기 위해 call을 한다. Explicit cursor는 이 두번째 call을 하지 않는다.

Implicit과 Explicit cursor에 대한 차이점과 사용법은 오라클의 PL/SQL교육 교재 또는 Developers Guide를 참조하기 바란다.

9. Stored Function을 활용한 SQL의 부하 감소

다음의 경우를 살펴 보자.

```
SELECT H.EMPNO, E.ENAME,
       H.HIST_TYPE, T.TYPE_DESC,
       COUNT(*)
  FROM HISTORY_TYPE T,
       EMP E,
       EMP_HISTORY H
 WHERE H.EMPNO = E.EMPNO
   AND H.HIST_TYPE = T.HIST_TYPE
 GROUP BY H.EMPNO, E.ENAME, H.HIST_TYPE, T.TYPE_DESC;
```

위 SQL문은 다음과 같은 FUNCTION을 만들어 사용할 경우 GROUP BY에 대한 부하를 감소시킬 수 있어 더 빠른 시간에 결과를 얻을 수 있다.

함수 1)
```
CREATE OR REPLACE FUNCTION Lookup_Hist_Type (typ IN NUMBER) RETURN VARCHAR2
AS
  tdesc VARCHAR2(30);

  CURSOR C1
  IS
    SELECT TYPE_DESC
      FROM HISTORY_TYPE
     WHERE HIST_TYPE = typ;
BEGIN
  OPEN C1;
  FETCH C1 INTO tdesc;
  CLOSE C1;

  RETURN (NVL(tdesc, '?');
END;
```

함수 2)
```
CREATE OR REPLACE FUNCTION Lookup_Emp (emp IN NUMBER) RETURN VARCHAR2
AS
  ename VARCHAR2(30);
  CURSOR C1 IS
    SELECT ENAME
      FROM EMP
     WHERE EMPNO = emp;
BEGIN
  OPEN C1;
  FETCH C1 INTO ename;
  CLOSE C1;

  RETURN (NVL(ename, '?');
END;
```

적용 예)
```
SELECT H.EMPNO, Lookup_Emp(H.EMPNO),
       H.HIST_TYPE, Lookup_Hist_Type(H.HIST_TYPE),
       COUNT(*)
  FROM EMP_HISTORY H
 GROUP BY H.EMPNO, H.HIST_TYPE;
```

10. Analytic Function의 활용

Analytic function이란 running summary, moving average, ranking, lead/lag comparisons 등 business 분야에서 자주 행하여지는 여러 가지 형태의 분석에 유용하게 활용될 수 있는 SQL function 들을 총칭한 개념이다.

이 함수들을 이용하면 ANSI SQL에서 여러 단계로 나누어 하던 작업들을 한번에 끝낼 수 있기

때문에 시스템에 부하를 적게 주면서도 개발 속도를 향상 시킬 수 있다.

Analytic function은 ORACLE 8.1.6 이후부터 사용 가능하고, 버전별로 사용 가능한 함수의 수가 다르므로 사용하는 시스템의 버전을 반드시 확인한 후 사용하여야 한다.

1) Analytic함수의 특징 및 장점

ㄱ) 특징

-. Analytic function은 SELECT 절과 ORDER BY 절에만 올 수 있다.
-. 실행계획 상에는 WINDOW SORT로서 표시된다.
-. analytic function을 적용한 후의 query의 결과집합 레벨은 analytic function을 적용하기 전의 결과집합 레벨과 동일하다. 즉, analytic function을 적용하기 전의 모든 로우(current row)에 대하여 analytic function result value가 존재한다.
-. Analytic function은 ORDER BY를 제외하고는 query에서 수행되는 가장 마지막 operation이다. 즉, 모든 join과 WHERE조건의 적용, GROUP BY & HAVING의 적용은 analytic function의 적용 전에 모두 완료된다. 그러므로, analytic function은 SELECT절과 ORDER BY절에만 사용될 수 있다.

ㄴ) 장점

-. Query speed의 향상
-. Self-join 또는 클라이언트 프로그램의 절차형 LOGIC으로 표현한 것을 native SQL에서 바로 적용할 수 있으므로 조인이나 클라이언트 프로그램의 overhead를 줄일 수 있음.
-. 향상된 개발 생산력(Enhanced Developer Productivity)
-. 개발자가 명백하고 간결한 SQL로 복잡한 분석작업을 수행할 수 있다.
-. 유지보수가 편하고 생산력을 높일 수 있다.
-. 배우기 쉬움 (Minimized Learning Effort)
-. 기존 SQL syntax를 그대로 따르기 때문에 이해가 쉽다.
-. 표준화(Standardized Syntax)
-. ANSI SQL로 채택되면 다양한 소프트웨어에 적용이 가능할 것이다.

2) 사용 방법

```
Analytic함수 ([arg1],[arg2],[arg3])
                OVER ([PARTITION BY value_expr] [ORDER BY … [WINDOWING …]]);
```

>> Analytic함수 : analytic function
>> arg1,2,3 : 각각의 analytic function은 0에서 3개의 arguments를 갖는다.
>> OVER절 : analytic함수의 대상이 되는 결과 집합에 대한 범위, 배열 순서 등에 대

해 기술한다.

3) 사용 예

EMP테이블에서 부서별 급여 순위와 급여 누적 합계, 그리고 회사 전체 순위와 급여 누적을 구해야 한다고 할 때, 기존의 방법과 analytic function을 이용한 두가지 방법을 살펴보면 다음과 같다.

예 1) Analytic function을 사용하지 않을 때

```
SELECT DEPTNO "부서명",
       ENAME  "성명",
       SAL    "급여",
       (SELECT COUNT(*)
          FROM SCOTT.EMP E3
         WHERE E3.DEPTNO = E0.DEPTNO
           AND (E3.SAL > E0.SAL
                OR (E3.SAL = E0.SAL
                    AND E3.ENAME <= E0.ENAME))) "부서내 급여 순서",
       (SELECT SUM(SAL)
          FROM SCOTT.EMP E2
         WHERE E2.DEPTNO = E0.DEPTNO
           AND (E2.SAL > E0.SAL
                OR (E2.SAL = E0.SAL
                    AND E2.ENAME <= E0.ENAME))) "부서내 급여 누적합계",
       (SELECT COUNT(*)
          FROM SCOTT.EMP E1
         WHERE E1.DEPTNO < E0.DEPTNO
            OR (E1.DEPTNO = E0.DEPTNO
                AND (E1.SAL > E0.SAL
                     OR (E1.SAL = E0.SAL
                         AND E1.ENAME <= E0.ENAME)))) "전체 순서",
       (SELECT SUM(SAL)
          FROM SCOTT.EMP E1
         WHERE E1.DEPTNO < E0.DEPTNO
            OR (E1.DEPTNO = E0.DEPTNO
                AND (E1.SAL > E0.SAL
                     OR (E1.SAL = E0.SAL
                         AND E1.ENAME <= E0.ENAME)))) "전체 급여 누적합계"
  FROM SCOTT.EMP E0
 ORDER BY DEPTNO, SAL DESC, ENAME;
```

예 2) Analytic function을 사용할 때

```
SELECT DEPTNO "부서명",
       ENAME  "성명",
       SAL    "급여",
       RANK() OVER (PARTITION BY DEPTNO
                    ORDER BY SAL DESC, ENAME)          "부서내 급여 순서",
       SUM(SAL) OVER (PARTITION BY DEPTNO
                      ORDER BY SAL DESC, ENAME)        "부서내 급여 누적합계",
```

```
       RANK() OVER (ORDER BY DEPTNO, SAL DESC, ENAME)   "전체 순서",
       SUM(SAL) OVER (ORDER BY DEPTNO, SAL DESC, ENAME) "전체 급여 누적합계"
  FROM SCOTT.EMP
 ORDER BY DEPTNO, SAL DESC;
```

ORACLE 9.2.0.2에서 사용 가능한 analytic function은 다음과 같다.

AVG *	CORR *	COVAR_POP *	COVAR_SAMP *	COUNT *
CUME_DIST	DENSE_RANK	FIRST	FIRST_VALUE *	LAG
LAST	LAST_VALUE *	LEAD	MAX *	MIN *
NTILE	PERCENT_RANK	PERCENTILE_CONT	PERCENTILE_DISC	RANK
RATIO_TO_REPORT	REGR_ (Linear Regression) Functions *	ROW_NUMBER	STDDEV *	STDDEV_POP *
STDDEV_SAMP *	SUM *	VAR_POP *	VAR_SAMP *	VARIANCE *

Asterisk(*)가 표시된 함수는 windowing_ 절을 포함한 모든 syntax를 사용할 수 있다.

V. TUNING

1. STATIC SQL의 활용

ORACLE에서의 SQL 수행과정을 간략하게 살펴보면, 크게 파싱(Parser), 옵티마이저(Query Optimizer), 로우소스 생성(Row Source Generator), SQL 실행(SQL Execution Engine)의 4단계로 분류할 수 있다. Parsing은 수행하고자 하는 SQL을 오라클 SGA(System Global Area) 내의 공유 SQL 영역(Shared Pool)에 넣고, 문법적 오류(syntax error)를 찾음과 동시에 Data Dictionary를 검색하여 SQL의 유효성을 확인한 후, 대상 테이블이나 인덱스 구조에 따른 실행계획(Execution Plan)을 작성하는 과정이다.

앞에서 말한 바와 같이 ORACLE은 SQL을 SGA에 공유함으로써 parsing 부하를 줄이는 방법을 사용하지만, Dynamic SQL을 사용하면 공유되지 않으므로 매번 새로 parsing하게 되어 시스템에 부하를 주게 된다. 이러한 부하를 줄이기 위해서는 Static SQL을 활용하여야 한다.

Dynamic SQL은 클라이언트 프로그램에서 조건에 따라 dynamic하게 SQL statement를 만들어 나가는 방식이며, Static SQL은 조건에 따라 유동적인 부분을 변수로 정의하여 SQL이 수행될 때 변수를 Binding함으로써 SQL을 공유하고 parsing에 따른 부하를 줄일 수 있다.

다음은 각 Tool별로 Static SQL을 사용하는 예이다. 일부분만을 추출한 것으로 이것만으로는 실행되지 않는 것도 있으므로 참고자료로만 활용하고, 자세한 사항은 각각의 매뉴얼을 참조하도록 한다.

1) Pro*C/C++

> Dynamic SQL

```
void getdata()
{
  char *dynstmt;

  printf("Wn Enter Employee Number to Query:");
  scanf("%s", emp_number);

  strcat(dynstmt, "SELECT ename INTO :emp_name FROM emp WHERE emp_no = ");
  strcat(dynstmt, emp_number);

  EXEC SQL dynstmt;

  printf("Wn Emplyoee %s :", emp_number);
  printf("Wn Name : %s", emp_name);
}
```

> Static SQL

```
void getdata()
{
  printf("Wn Enter Employee Number to Query:");
  scanf("%dWn", emp_number);

  EXEC SQL SELECT ename INTO :emp_name
             FROM emp
            WHERE emp_no = :emp_number;

  printf("Wn Emplyoee %d :", emp_number);
  printf("Wn Name : %s", emp_name);
}
```

2) JAVA

> Dynamic SQL

```
try {
  String url = "jdbc:myprotocol:mydatabase";
  Connection db = DriverManager.getConnection(url, "myid", "mypassword");

  Statement stmt = db.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT * FROM EMP WHERE DEPTNO=" + pDeptNo);
            ...
} catch (SQLException e) {
  System.err.println("SQL Error : " + e.getMessage());
}
```

> Static SQL

```
try {
  String url = "jdbc:myprotocol:mydatabase";
  Connection db = DriverManager.getConnection(url, "myid", "mypassword");

  PreparedStatement stmt = db.prepareStatement(
                           "SELECT * FROM EMP WHERE DEPTNO = ?");
  stmt.setString(1, "10");
  ResultSet rs = stmt.executeQuery();
            ...
} catch (SQLException e) {
  System.err.println("SQL Error : " + e.getMessage());
}
```

3) DELPHI

> Dynamic SQL

```
With Qeury1 do begin
  Close;
  Sql1.CLEAR;
```

```
  Sql1.ADD('SELECT * FROM EMP WHERE DEPTNO = ' + '10');
  Open;
End;
```

> Static SQL

```
With Qeury1 do begin
  Close;
  Sql1.CLEAR;
  Sql1.ADD('SELECT * FROM EMP WHERE DEPTNO = :pDeptNo');
  ParamByName('pDeptno').AsNumber := eDeptNo.Text;
  Open;
End;
```

4) VB

> Dynamic SQL

```
Set RS = New ADODB.Recordset
RS.ActiveConnection = ADOConn
RS.CursorLocation = adUseClientBatch

sSql = "SELECT * FROM EMP " & _
       " WHERE DEPTNO = '" & pDeptNo & "'"

RS.Open sSql, , adOpenKeyset, adLockBatchOptimistic, adCmdUnspecified
```

> Static SQL

```
OraDatabase.Parameters.Add "pDeptno", 10, ORAPARAM_INPUT
OraDatabase.Parameters("pDeptNo").ServerType = ORATYPE_NUMBER

Dim OraDynaset As Object
Set OraDynaset = OraDatabase.CreateDynaset( _
                 "SELECT * FROM EMP WHERE DEPTNO = :pDeptNo", ORADYN_ORAMODE)
```

5) PHP

> Dynamic SQL

```
$sql = "select * from emp where deptno = ".$dno;
$stmt = OCIParse($conn, $sql);
OCIExecute($stmt);
$nrows = OCIFetchStatement($stmt, $results);
```

> Static SQL

```
$sql = "select * from emp where deptno = :pDeptNo";
$stmt = OCIParse($conn, $sql);
OCIBindByName($stmt,":pDeptNo",&$dno,32);
OCIExecute($stmt);
$nrows = OCIFetchStatement($stmt, $results);
```

2. 비효율적인 SQL을 확인하는 방법

아래의 SQL문을 이용하여 비효율적인 SQL문을 확인해 볼 수 있다.

```
SELECT EXECUTIONS, DISK_READS, BUFFER_GETS,
       ROUND((BUFFER_GETS-DISK_READS)/BUFFER_GETS,2) Hit_Ratio,
       ROUND(DISK_READS/EXECUTIONS,2) Reads_Per_Run,
       SQL_TEXT
  FROM V$SQLAREA
 WHERE EXECUTIONS > 0
   AND BUFFER_GETS > 0
   AND (BUFFER_GETS - DISK_READS) / BUFFER_GETS < 0.80
 ORDER BY 4 DESC;
```

위의 문장은 Buffer Cache에 대한 Hit ratio가 80% 미만인 비효율적인 SQL문을 찾아내는 SQL이다.

DBA 또는 system view를 볼 수 있는 권한을 가져야만 실행할 수 있으므로 주의하도록 한다.

참고 - Buffer Cache의 기본적인 기능은 여러 프로세스에 의해 공통으로 자주 액세스 되는 데이터베이스 블록을 메모리에 cache하여 물리적인 디스크 IO를 최소화함으로써 더 빠른 액세스 속도를 제공하기 위한 것이다.

버퍼캐쉬의 적중률(Hit ratio)이란 어플리케이션이 액세스한 메모리 block 가운데 이미 cache가 되어 있어 물리적 I/O 없이 액세스 할 수 있었던 block의 비율을 나타낸다.

만약 데이터베이스 버퍼의 적중률(Hit ratio)이 권장치(90%) 미만일 경우에는 할당된 버퍼 캐쉬의 크기가 너무 적거나, 또는 지나치게 많은 I/O를 유발하는 어플리케이션이 존재한다는 것을 의미한다.

3. TKPROF를 이용하여 성능 통계정보 보기

SQL TRACE FACILITY는 SQL문 사용에 대한 성능을 분석하기 위해서 사용된다. 이러한 SQL TRACE FACILITY를 이용하면 각각의 SQL문에 대해서 다음과 같은 정보를 얻을 수 있다.

> parse, execute, fetch count
> CPU 와 elapsed 시간
> physical reads 와 logical reads
> 처리된 row 의 수

SQL TRACE FACILITY 는 SESSION 혹은 INSTANCE 단위로 할 수 있고 TRACE 결과 파일은 tkprof UTILITY에 의해 사용자가 읽을 수 있는 형태로 변환시킨다.

▶ SQL Trace 사용법

1) SQL TRACE enable 및 TRACE 파일 디렉토리 지정

ㄱ) INSTANCE 단위

init.ora 파일에 다음 두개의 PARAMETER를 추가하고 DATABASE를 다시 STARTUP 시킨다.

```
sql_trace = true
timed_statistics = true
```

timed_statistics는 시스템에 많은 LOAD가 걸리므로 전체 INSTANCE 단위에 사용하는 것은 바람직 하지 않다.

ㄴ) SESSION 단위

>> SQL*PLUS

```
$ sqlplus scott/tiger
SQL> ALTER SESSION SET SQL_TRACE = TRUE;
SQL> sql문장 실행
SQL> exit
```

>> PRO*C

```
EXEC SQL CONNECT :username;
EXEC SQL ALTER SESSION SET SQL_TRACE = TRUE;
```

이렇게 하면 user_dump_dest directory에 trace file이 생성된다. user_dump_dest가 어디로 지정되어 있는지는 다음과 같이 확인한다.

```
SQL> select value from v$parameter where name = 'user_dump_dest';
```

2) TRACE 파일 변환

SQL문을 실행하면 user_dump_dest에 지정된 디렉토리에 TRACE 파일이 생기고 tkprof를 이용하여 파일을 변환시킨다. TRACE 파일은 쉽게 찾을 수 있는 형태가 아니므로 SQL 문을 실행하기 전에 dump 디렉토리에 있는 ora_xxxx.trc 파일을 모두 삭제하거나 가장 최근에 생긴 파일 중에서 찾아야 한다.

이 때 해당 user에 plan_table 이라는 table이 없으면 utlxplan.sql을 수행하여 table을 만든다.

예) $ cd $ORACLE_HOME/rdbms/log

$ tkprof ora_xxx.trc result.out sort=fchqry,fchcu explain=scott/tiger print=20

ora_xxx.trc : TRACE 파일명
result.out : 결과 파일명
sort : 지정된 OPION(fchqry, fchcu)에 ASCENDING순으로 SQL문을 SORTING 한다.
explain : SQL 문의 EXECUTION PLAN 을 발생시킨다.
print : 지정된 개수의 SQL문에 대해서만 TRACE 결과를 PRINT 한다.

3) SQL TRACE 결과 분석

```
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************

SELECT COUNT(*)
FROM EMP E, DEPT D
WHERE E.DEPTNO=D.DEPTNO

call     count  cpu    elapsed   disk    query    current    rows
------------------------------------------------------------------
Parse    2      0.00    0.00       0       0        0          0
Execute  2      0.00    0.00       0       0        0          0
Fetch    2      0.00    0.00       0      33        2          1

Misses in library cache during parse: 1
Parsing user id: 8  (SCOTT)
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
      0    SORT (AGGREGATE)
     16      NESTED LOOPS
     16        TABLE ACCESS (FULL) OF 'EMP'
     16        INDEX (UNIQUE SCAN) OF 'DEPT_PRIMARY_KEY' (UNIQUE)
********************************************************************************
```

▶ 분석 방법

a. CPU, elapsed 정보가 없는 경우는 init.ora에 timed_statistics 설정을 확인한다.

b. Execute count와 Fetch count가 동일하게 크다고 하면 ARRAY FETCH 사용을 고려함.

c. fetch된 rows 수 : query + current = 1 : 4 이하이면 SQL 문은 적절히 사용된 경우이고, row 수에 비하여 query + count가 상당히 크면 부적절하게 사용된 SQL 문이므로(count,

sum, distinct 등 GROUP function을 사용하는 경우는 예외) 다음 내용들에 대해서 재검토가 필요하다.

>> INDEX 사용, 구성 여부

>> ROWID 사용

>> COST_BASED OPTIMIZER 사용(ORACLE 7)

>> ARRAY FETCH 사용

>> SORTING을 피할 수 있는 SQL 문 구사

d. Parse count, Execute count가 비슷한 경우 RELEASE_CURSOR, HOLE_CURSOR OPTION 사용하여 Parse count를 줄임.

4. SQL문 분석을 위한 EXPLAIN PLAN의 활용

Explain Plan은 SQL문을 실행하지 않고 실행되어질 경로를 파악함으로써 성능을 가늠해 볼 수 있도록 ORACLE에서 제공하는 기능이다. Explain Plan의 결과에서는 SQL문을 수행하기 위해 ORACLE이 하게 될 테이블의 조회 또는 join 방법, index search 또는 full table scan과 같은 access의 순서 및 형태, 그리고 사용하게 될 index의 이름 등에 대한 내용 등을 알 수 있다.

해석하는 방법은 맨 안쪽부터 바깥 쪽으로, 그리고 위에서부터 아래 쪽으로 읽어 나가야 한다. 그러므로 만약 두개의 operation이 나열 될 경우 일반적으로 더 들쭉날쭉한 문장이 먼저 실행되고, 둘 다 동일한 레벨이라면 먼저 기술 된 문장이 먼저 실행된다.

가장 대표적인 Access Type은 NESTED LOOPS, HASH join, MERGE join이다.

NESTED LOOPS은 **'맨 안쪽부터 바깥 쪽으로, 그리고 위에서부터 아래 쪽으로'**의 룰에 따라 건별로 순차적으로 데이터를 읽어 나간다.

MERGE join 방식은 두 테이블을 읽어 sort한 후 서로 값을 비교해 나가는 방식이며, HASH join 방식은 데이터를 읽되 내부적으로 hash function을 이용하여 정렬한 후 이를 통해 join해 나가는 방식이다.

어느 한가지 방식이 항상 유리한 것은 아니고, 데이터의 건수나 인덱스의 구성, 통계정보의 유무에 따라 성능이 결정된다.

이에 대해 깊게 배우고자 하는 분은 ORACLE 기술 문서를 참조하기 바란다.

다음은 자주 볼 수 있는 OPERATION과 OPTION을 정리한 표이다.

표) EXPLAIN PLAN에 의해 자주 생성되는 OPERATION과 OPTION

Operation	Option	Description
AND-EQUAL		단일 컬럼 인덱스에 대한 access 시 사용하며, 인덱스 merge 를 통해 rowid 별로 중복을 제거 한다.
CONNECT BY		CONNECT BY 절에 의해 순차적인 결과값을 보여준다.
CONCATENATION		여러 결과 집합을 union-all 하여 하나의 집합으로 돌려 준다.
COUNT		조건에 맞는 집합 내의 row 건수를 돌려 준다.
	STOPKEY	ROWNUM 에 의해 where 절에서 건수를 세서 중지시킨다.
FILTER		여러 조건에 의해 결과집합을 걸러서 돌려 준다.
FIRST ROW		Query 에 의해 선택된 row 중 처음 row 만 돌려 준다.
FOR UPDATE		FOR UPDATE 절에 의해 선택되고 update 를 위해 lock 이 걸려 있을 때
HASH JOIN +		DSS 나 Batch 에서 다량의 데이터에 대해 사용할 때 유용하다. CBO 는 메모리상에서 join key 를 이용해 hash 테이블을 생성하고 이를 통해 각 테이블에 access 한다.
	ANTI	Hash anti-join.
	SEMI	Hash semi-join.
INDEX *	UNIQUE SCAN	Unique-index 를 통해 테이블의 row 에 access 한다.
	RANGE SCAN	Non-unique index 를 통하거나 unique index 의 특정 범위에 대한 테이블의 row 에 access 할 때
	FULL SCAN	인덱스에 대해서 full scan 하여 결과값을 보여준다.
	FAST FULL SCAN	인덱스를 multiblock 을 읽어 결과값을 보여준다. CBO 에서만 사용 가능하다.
	SKIP SCAN	결합인덱스에서 선행 컬럼을 건너뛰고 scan 하여 결과값을 보여준다. CBO 에서만 사용 가능하다.
INTERSECTION		교집합 추출. 중복값 없음.
MERGE JOIN +		먼저 자신의 조건만으로 액세스한 후 각각을 소트하여 merge 해 가는 조인.
	OUTER	Merge join 에 outer join 이 지정된 경우
	ANTI	Merge anti-join.
	SEMI	Merge semi-join.
	CARTESIAN	각 결과 집합을 이용해 Cartesian product 생성
MINUS		Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.
NESTED LOOPS +		먼저 어떤 DRIVING 테이블의 ROW를 ACCESS한 후 그 결과를 이용해 다른 테이블을 연결하는 조인
	OUTER	위와 동일한 방법으로 OUTER JOIN을 한다.
REMOTE		분산 DATABASE 에 있는 객체를 추출하기 위해 DATABASE LINK 를 사용하는 경우.
SORT	AGGREGATE	그룹함수(SUM, COUNT 등)를 사용하여 하나의 ROW를 추출
	UNIQUE	같은 ROW를 제거하기 위한 소트
	GROUP BY	액세스 결과를 GROUP BY하기 위한 소트
	JOIN	MERGE JOIN을 하기 위한 소트
	ORDER BY	ORDER BY를 위한 소트

TABLE ACCESS *	FULL	테이블의 전체 row 에 대해 access 한다.
	CLUSTER	Indexed cluster key 를 통해 테이블의 row 에 access 한다.
	HASH	Hash cluster key 를 통해 테이블의 row 에 access 한다.
	BY ROWID RANGE	Rowid 의 범위에 의해 테이블의 row 에 access 한다.
	BY USER ROWID	사용자가 지정한 rowid 를 통해 테이블의 row 에 access 한다.
	BY INDEX ROWID	Partition 되지 않은 index 를 통해 테이블의 row 에 access 한다.
	BY GLOBAL INDEX ROWID	Partition 된 테이블에 대해 global index 를 통해 테이블의 row 에 access 한다.
	BY LOCAL INDEX ROWID	Partition 된 테이블에 대해 local index 를 통해 테이블의 row 에 access 한다.
UNION		중복 없는 합집합을 구한다. 항상 전체 범위 처리
VIEW		어떤 처리에 의해 생성되는 뷰에서 추출한다. 주로 sub-query에 의해서 수행된 결과

* access methods + join operations

위 표)는 자주 사용하는 것들을 정리한 것이므로, 그 외에 더 자세한 내용을 알고자 하면 Oracle9i Database Performance Tuning Guide and Reference를 참조하기 바란다.

5. HINT의 사용

HINT란 ORACLE Optimizer가 실행 경로를 생성할 때 그 경로를 제어하기 위해 사용자가 제시하는 가이드라인이다. 가이드라인이라 한 이유는 HINT를 제시한다고 해서 경로 생성이 반드시 원한대로 되는 것이 아니라는 것이다.

아래는 주로 사용하는 아래의 힌트들에 대해 설명하고자 한다.

ALL_ROWS, FIRST_ROWS, CHOOSE, RULE, FULL, ROWID, USE_NL, USE_MERGE, USE_HASH, INDEX, INDEX_ASC, INDEX_DESC

예) OPTIMIZER_MODE=CHOOSE (init.ora)

TABLE : SCOTT.EMP, SCOTT.DEPT

INDEX : EMP - EMPNO에 PK emp_pk, DEPTNO에 인덱스 emp_idx01

DEPT - DEPTNO에 PK dept_pk

아래의 analyze 문을 수행한다.

```
ANALYZE TABLE SCOTT.EMP  COMPUTE STATISTICS;
ANALYZE TABLE SCOTT.DEPT COMPUTE STATISTICS;
```

1) ALL_ROWS

ALL_ROWS는 대상 결과를 모두 표시하는데 가장 좋은 경로를 생성하고자 할 때 사용한다.

ALL_ROWS를 할 경우 Full table scan을 선호하며 CBO는 default로 ALL_ROWS를 선택한다. 그러나 아래와 같이 유용한 인덱스가 있거나 다른 결과의 통계정보가 있는 경우 아래와 같이 INDEX SCAN을 한다.

```
SQL> SELECT /*+ ALL_ROWS */
            EMPNO, ENAME
       FROM EMP
      WHERE EMPNO = 7566;

     Execution Plan
     ----------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=HINT: ALL_ROWS (Cost=1 Card=4 Bytes=348)
     1  0  TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.EMP' (Cost=1 Card=4 Bytes=348)
     2  1    INDEX (RANGE SCAN) OF 'SCOTT.EMP_PK' (UNIQUE) (Cost=1 Card=2)
```

2) FIRST_ROWS

FIRST_ROWS는 대상 결과 중 첫 row를 표시하는데 가장 좋은 경로를 생성하고자 할 때 사용한다.

Full table scan보다는 index scan을 선호하며 interactive application인 경우 best response time을 제공한다. 또한 sort merge join보다는 nested loop join을 선호한다.

```
SQL> SELECT /*+ FIRST_ROWS */
            E.EMPNO, E.ENAME, D.DNAME
       FROM EMP E,
            DEPT D
      WHERE E.DEPTNO = D.DEPTNO

     Execution Plan
     ----------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=411 Card=409 Bytes=47K)
     1  0  NESTED LOOPS (Cost=411 Card=409 Bytes=47K)
     2  1    TABLE ACCESS (FULL) OF 'SCOTT.EMP' (Cost=2 Card=409 Bytes=35K)
     3  1    TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.DEPT' (Cost=1 Card=1 Bytes=30)
     4  3     INDEX (UNIQUE SCAN) OF 'SCOTT.DEPT_PK' (UNIQUE) (Card=1)
```

또, select list에 따라 index scan을 하는 table이 바뀔 수도 있다.

```
SQL> SELECT /*+ FIRST_ROWS */
            D.DEPTNO, D.DNAME
       FROM EMP E,
            DEPT D
      WHERE E.DEPTNO = D.DEPTNO
```

```
Execution Plan
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=411 Card=409 Bytes=14K)
1  0  NESTED LOOPS (Cost=411 Card=409 Bytes=14K)
2  1    TABLE ACCESS (FULL) OF 'SCOTT.DEPT' (Cost=2 Card=409 Bytes=9K)
3  1    INDEX (RANGE SCAN) OF 'SCOTT.EMP_IDX01' (NON-UNIQUE)
        (Cost=1 Card=1 Bytes=13)
```

Group by 가 있는 SQL문은 FIRST_RIWS가 있다 하더라도 index scan을 하지 않는다.

```
SQL> SELECT /*+ FIRST_ROWS */
            COUNT(*)
       FROM EMP E
      GROUP BY E.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=18 Card=409 Bytes=5K)
1  0  SORT (GROUP BY) (Cost=18 Card=409 Bytes=5K)
2  1    TABLE ACCESS (FULL) OF 'SCOTT.EMP' (Cost=2 Card=409 Bytes=5K)
```

3) CHOOSE

Hint level의 CHOOSE는 RBO인지 CBO인지를 선택한다. 만약 주어진 table의 통계 정보가 없다면 RBO를 사용한다.

4) RULE

RBO를 사용하도록 지정한다.

```
SQL> SELECT /*+ RULE */
            COUNT(*)
       FROM EMP E
      GROUP BY E.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=HINT: RULE
1  0  SORT (GROUP BY)
2  1    TABLE ACCESS (FULL) OF 'SCOTT.EMP'
```

5) FULL

FULL 힌트는 참조 테이블에 대해 full table scan을 하도록 ORACLE optimizer를 유도한다. 이는 Index가 있지만 선택도(selectivity)가 좋지 않은 경우 full table scan을 선택하도록 한다.

```
SQL> SELECT /*+ FULL(EMP) */
            EMPNO, ENAME
       FROM EMP
      WHERE EMPNO = 10;

     Execution Plan
     ----------------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=32)
     1  0  TABLE ACCESS (FULL) OF 'SCOTT.EMP' (Cost=2 Card=1 Bytes=32)
```

6) ROWID

ROWID 힌트는 optimizer로 하여금 테이블의 row에 대해 TABLE ACCESS BY ROWID 방식으로 접근하도록 유도한다.

일반적으로 TABLE ACCESS BY ROWID 방식은 큰 테이블에서 다른 방식보다 빠르게 결과값을 보여 준다. 이 방식을 적용하여 query를 수행하고자 할 때에는 ROWID를 알고 있거나 index를 사용하여야 한다.

```
SQL> SELECT /*+ ROWID(EMP) */
            *
       FROM EMP
      WHERE EMPNO = 7935;

     Execution Plan
     ----------------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=32)
     1  0  TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.EMP' (Cost=1 Card=1 Bytes=32)
     2  1   INDEX (UNIQUE SCAN) OF 'SCOTT.EMP_PK' (UNIQUE) (Card=14)
```

7) USE_NL, USE_HASH, USE_MERGE

USE_NL은 Nested Loop 방식으로 테이블을 join하게 한다. Nested Loop Join은 하나의 테이블의 처리 범위를 하나씩 액세스하면서 그 추출된 값으로 다른 테이블의 값을 찾아 나가는 방식이다.

```
SQL> SELECT /*+ USE_NL(E D) */
            E.EMPNO, E.ENAME, D.DNAME
       FROM EMP E,
            DEPT D
      WHERE E.DEPTNO = D.DEPTNO;

     Execution Plan
     ----------------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=6 Card=14 Bytes=294)
     1  0  TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.EMP' (Cost=1 Card=4 Bytes=40)
     2  1   NESTED LOOPS (Cost=6 Card=14 Bytes=294)
     3  2    TABLE ACCESS (FULL) OF 'SCOTT.DEPT' (Cost=2 Card=4 Bytes=44)
     4  2    INDEX (RANGE SCAN) OF 'SCOTT.EMP_IDX01' (NON-UNIQUE) (Card=5)
```

USE_MERGE는 Sort Merge 방식으로 테이블을 join하게 만든다. Sort Merge Join은 join하고자 하는 테이블을 각각 처리한 결과를 join key로 정렬하여 상호 비교하는 방식을 말한다.

```
SQL> SELECT /*+ USE_MERGE(E D) */
            E.EMPNO, E.ENAME, D.DNAME
       FROM EMP E,
            DEPT D
      WHERE E.DEPTNO = D.DEPTNO;

     Execution Plan
     ----------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=20 Card=14 Bytes=294)
     1  0  MERGE JOIN (Cost=20 Card=14 Bytes=294)
     2  1    TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.DEPT' (Cost=2 Card=4 Bytes=44)
     3  2     INDEX (FULL SCAN) OF 'SCOTT.DEPT_PK' (UNIQUE) (Cost=1 Card=4)
     4  1    SORT (JOIN) (Cost=18 Card=14 Bytes=140)
     5  4     TABLE ACCESS (FULL) OF 'SCOTT.EMP' (Cost=2 Card=14 Bytes=140)
```

USE_HASH는 hash function을 이용하여 join하는 방식이다. 데이터를 읽되 내부적으로 hash function을 이용하여 메모리에 정렬한 후 이를 통해 join해 나가는 방식이다.

```
SQL> SELECT /*+ USE_HASH(E D) */
            E.EMPNO, E.ENAME, D.DNAME
       FROM EMP E,
            DEPT D
      WHERE E.DEPTNO = D.DEPTNO;

     Execution Plan
     ----------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=294)
     1  0  HASH JOIN (Cost=5 Card=14 Bytes=294)
     2  1    TABLE ACCESS (FULL) OF 'SCOTT.DEPT' (Cost=2 Card=4 Bytes=44)
     3  1    TABLE ACCESS (FULL) OF 'SCOTT.EMP' (Cost=2 Card=14 Bytes=140)
```

8) INDEX, INDEX_ASC

INDEX 힌트는 지정한 테이블에 대해 인덱스를 통한 scan을 하도록 optimizer를 유도한다. INDEX 힌트를 사용할 때에는 인덱스명을 지정하지 않아도 되나, 특정 인덱스를 사용하고자 할 때에는 인덱스명을 추가할 수도 있다. Default로 index scan은 오름차순이다.

```
SQL> SELECT /*+ INDEX(EMP EMP_PK) */
            *
       FROM EMP
      WHERE EMPNO = 7935;

     Execution Plan
     ----------------------------------------------------------------------------
     0    SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=32)
     1  0  TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.EMP' (Cost=1 Card=1 Bytes=32)
     2  1    INDEX (UNIQUE SCAN) OF 'SCOTT.EMP_PK' (UNIQUE) (Card=14)
```

9) INDEX_DESC

지정된 index를 내림차순으로 쓰도록 지정한다.

```
SQL> SELECT /*+ INDEX_DESC(EMP EMP_PK) */
            *
       FROM EMP
      WHERE EMPNO = 7935;
Execution Plan
----------------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=32)
1  0  TABLE ACCESS (BY INDEX ROWID) OF 'SCOTT.EMP' (Cost=1 Card=1 Bytes=32)
2  1   INDEX (RANGE SCAN DESCENDING) OF 'SCOTT.EMP_PK' (UNIQUE) (Card=1)
```

다음은 ORACLE에서 사용되는 힌트를 정리한 표이다.

표) ORACLE Hint

분류	HINT
Optimization Goals and Approaches	ALL_ROWS and FIRST_ROWS
	CHOOSE
	RULE
Access Method Hints	AND_EQUAL
	CLUSTER
	FULL
	HASH
	INDEX and NO_INDEX
	INDEX_ASC and INDEX_DESC
	INDEX_COMBINE
	INDEX_FFS
	ROWID
Join Order Hints	ORDERED
	STAR
Join Operation Hints	DRIVING_SITE
	HASH_SJ, MERGE_SJ, and NL_SJ LEADING
	USE_HASH and USE_MERGE
	USE_NL
Parallel Execution Hints	PARALLEL and NOPARALLEL
	PARALLEL_INDEX
	PQ_DISTRIBUTE
	NOPARALLEL_INDEX
Query Transformation Hints	EXPAND_GSET_TO_UNION
	FACT and NOFACT
	MERGE
	NO_EXPAND
	NO_MERGE

	REWRITE and NOREWRITE
	STAR_TRANSFORMATION
	USE_CONCAT
Other Hints	APPEND and NOAPPEND
	CACHE and NOCACHE
	CURSOR_SHARING_EXACT
	DYNAMIC_SAMPLING
	NESTED_TABLE_GET_REFS
	UNNEST and NO_UNNEST
	ORDERED_PREDICATES
	PUSH_PRED and NO_PUSH_PRED
	PUSH_SUBQ

각 힌트의 기능에 대한 설명을 보고자 할 때에는 Oracle9i SQL Reference Relase2(9.2)를 참고하기 바란다.

6. Export와 Import의 성능 향상

Oracle은 논리적 데이터 복사 방법으로 Export/Import를 지원한다. 데이터의 추출을 위해서 Export를, 데이터를 Load하기 위해서 Import를 사용한다.

이들을 이용하여 작업할 때 큰 사이즈의 buffer를 지정한 후 Export나 Import를 수행하면 작업 시간을 단축할 수 있다.

가령 10MB의 buffer를 지정하게 되면 ORACLE은 지정한 크기 내에서 가능한 크게 buffer를 확보하게 되지만 그보다 작다고 해서 에러를 내지는 않는다.
그러나 만약 테이블의 가장 큰 컬럼보다 작은 사이즈를 지정하게 되면 이 값은 무시된다.

> DIRECT EXPORT의 활용

Export툴의 경우 추출 메커니즘에서 2가지 방식을 지원하는데 Conventional Path방식과 Direct Path방식이 그것이다.
Direct로 Export를 수행하면 Buffer Cache와 Evaluating Buffer를 수행하지 않음으로써 보다 빨리 Export를 할 수 있다. 또한 Instance에서 다른 자원과 경합이 발생하지 않음으로 효율적이다. 일반적으로 30에서 50% 이상의 속도 향상을 이룰 수 있다

Direct방식은 성능면에서는 효율적이지만 아래와 같이 주의할 점이 있다.

- Direct-path 옵션은 대화식으로 사용될 수 없음. (direct=y)
- Client-side와 server-side 의 character set이 일치해야 함.

- BUFFER parameter는 영향을 끼치지 않음.
- LOB, BFILE, REF, 또는 객체 type을 포함한 row에 대해 사용할 수 없음

위와 같은 제약조건을 만족한다면 당연히 Direct를 사용하는 것이 유리하다.
Interactive 방식으로 처리할 때 DIRECT=Y를 지정하여 사용하면 된다.

단, Direct 방식을 사용할 때에는 RECOREDLENGTH parameter를 OS의 I/O 단위와 Oracle의 DB_BLOCK_SIZE의 배수로 것이 속도향상에 도움이 된다.

7. 물리적 I/O의 분산

일반적으로 사용자용 테이블과 인덱스를 위한 테이블스페이스는 논리적으로나 물리적으로 분리하여 지정하는 것이 좋다. 또한 ORACLE 시스템을 위한 테이블스페이스는 절대로 사용하지 않도록 한다.

그 이유는 하나의 디스크의 헤드가 인덱스의 정보를 읽는 동안 다른 디스크의 헤드는 테이블의 데이터를 읽기 위한 위치에 가 있을 수 있다. 읽기와 쓰기가 하나의 디스크에서 일어나면 디스크의 헤드가 번갈아 인덱스와 테이블 위치를 오가며 읽어야 하기 때문에 I/O Contention이 일어나서 수행 속도에 지장을 주게 된다.

그러나 근래에 와서는 STORAGE가 대용량화, 고성능화 되어 가고 SAN 또는 NAS 환경으로 구성되면서 DISK 단위의 I/O에 대한 경합은 크게 중요하지 않게 되어 가고 있다.

8. CPU Tuning

오라클 instance를 background process + SGA라고 말할 때 처음 instance가 기동되면 SGA가 할당된다. 이 때 이 SGA를 구성하는 메모리 영역은 크게 3부분으로 구성된다고 할 수 있다.

> Shared pool
> Redo log buffer
> DB buffer cache.

여기서는 SGA를 구성하는 shared pool의 size를 시스템에 맞게 산출하는 방법에 대해서 알아본다.

Shared pool 영역 구성은 MTS 방식으로 접속된 session의 PGA, Shared SQL area 그리고 dynamic하게 할당되는 data structure로 구성된다.

1) Shared Pool

Shared pool은 ?/dbs/initSID.ora(parameter file)의 SHARED_POOL_SIZE라는 parameter로 그 크기를 지정하는데 default값은 3.5M를 갖게 된다.

일반적으로 shared pool이 얼마나 사용되는가 하는 문제는 application dependent하므로 각 application을 조사해야 한다.

시스템에서 필요로 하는 크기를 검사하기 위해 아주 큰 크기로 parameter file에 지정하여 dynamic하게 할당되는 SGA가 충분히 큰 값을 갖게 한 후, 검사가 끝난 다음 아래에서 계산된 size로 변경 해 주도록 한다.

2) 계산 공식

```
Session당 최대 메모리 사용량(Max Session Memory) *  동시 접속하는 User의 수
+ Shared SQL 영역으로 사용되는 메모리 양
+ Shared PLSQL을 위해 사용하는 메모리 영역
+ 최소 30%의 여유 공간
= Minimum Shared Pool
```

3) 계산 예제

ㄱ) 적당한 user session에 대한 session id를 찾는다.

```
SQL> SELECT SID
       FROM V$PROCESS P,
            V$SESSION S
      WHERE P.ADDR = S.PADDR
        AND S.USERNAME = 'SCOTT';

       SID
----------
        29
1 rows selected.
```

ㄴ) 이 session id에 대한 maximum session memory를 찾는다.

```
SQL> SELECT VALUE
       FROM V$SESSTAT S,
            V$STATNAME N
      WHERE S.STATISTIC# = N.STATISTIC#
        AND N.NAME = 'SESSION UGA MEMORY MAX'
        AND SID = 29;

  VALUE
-----------
    273877
1 rows selected.
```

ㄷ) Total shared SQL area를 구한다.

```
SQL> SELECT SUM(SHARABLE_MEM)
       FROM V$SQLAREA;


 SUM(SHARAB
 ------------------
           8936625
 1 row selected.
```

ㄹ) PLSQL sharable memory area를 구한다.

```
SQL> SELECT SUM(SHARABLE_MEM)
       FROM V$DB_OBJECT_CACHE;

 SUM(SHARAB
 ------------------
           4823537
 1 row selected.
```

ㅁ) Shared pool size를 계산한다.

```
274K shared memory * 400 users
+ 9M Shared SQL Area
+ 5M PLSQL Sharable Memory
+ 60M Free Space (30%)
= 184M Shared Pool
```

이 예제에서는 Shared pool의 size는 184M가 적당하다고 할 수 있다.

4) Shared Memory 부족(ORA-4031)에 대한 대처

Ora-4031이 발생하는 원인은 2가지이다. 첫째 절대적으로 shared pool size가 작아서 나는 경우와, 둘째로 memory는 있으나 적재하려 하는 PL/SQL package가 너무 커서 연속된 shared pool영역을 점유하지 못하는 경우가 있다.

만일 첫번의 경우라면 적당한 계산 과정으로 계산하여 parameter file에서 SHARED_POOL_SIZE를 늘려주고, 두 번째 경우라면 다음과 같은 방법으로 에러를 피해 갈 수 있다.

9. STATSPACK를 활용한 성능 분석

StatsPack은 일련의 SQL, PL/SQL 및 SQL*Plus 스크립트로 구성되어 있으며, 사용자가 각종 성능 관련 데이터를 수집, 저장, 출력 및 이런 일련의 과정을 자동화할 수 있게 해 준다.

DB 계정은 설치 script를 실행할 때 자동으로 생성된다. (PERFSTAT 계정). PERFSTAT 계정은 성능 튜닝에 필요한 v$view에 대해 제한적으로 query-only 권한을 부여 받게 된다.

StatsPack은 그 동안 널리 사용되어 오던 UTLBSTAT/UTLESTAT 튜닝 스크립트와는 근본적으로 다르며, 더 많은 정보를 수집하고, 성능 관련 통계정보를 데이터베이스에 축적한다는 장점이 있다. 축적된 성능 관련 정보는 후에도 보고서 작성 및 분석을 위해 다시 사용될 수 있다.

수집된 데이터는 제공된 보고서에 의해 분석되는데, 보고서에서는 "instance health and load" 라는 종합 보고서를 포함해, 자원을 많이 사용하는 SQL 문장 및, wait event, init 파라미터 등 다양한 정보가 포함된다.

기타 설치 및 활용법은 OTN에 있는 자료를 참조하기 바란다.

(http://211.106.111.2:8880/bulletin/list.jsp?seq=17118&pg=0&sort_by=last_updated&keyfield=subject&keyword=statspack)

- 참고 자료 -

1. 대용량 데이터베이스 솔루션 I, II / ㈜엔코아정보컨설팅 / 이화식, 조광원 공저
2. SQL튜닝 Review 및 실무 / ㈜웨어밸리 / 2001
3. Digital Contetns / 한국데이터베이스진흥센터 / 이현호 / 2002.01
4. 오라클 옵티마이저의 기본 원리 / 한국오라클 / 이상호 / 2002
5. Technical Column / OTN (Oracle Technical Network), ORACLE / Robin Schumacher
6. How To Write Efficient SQL Queries with Tips N Tricks / Prashant S.Sarode
7. Oracle9*i* Database Performance Tuning Guide and Reference Release 2 (9.2) / March 2002
8. Oracle9i SQL Reference Release 2 (9.2) / March 2002.