

Oracle Optimizer의 원리 이해 및 SQL & 애플리케이션의 튜닝 (하)

오라클 튜닝 기법의 100% 활용

글 | 최세훈 (한국오라클 DB Tech팀) sehoon.choi@oracle.com

지난 회에서는 튜닝에 들어가기 위해 먼저 Oracle Optimizer의 원리와 특징에 대해서 설명했다.

이번 회에서는 조인 메소드별 특징과 플랜 보는 법을 이해하고, 실제 오라클에서 제공하는 튜닝 기법들을 활용해 보도록 하자.

숲을 보는 튜닝

튜닝에는 정답이 없다. 즉 튜닝은 시스템의 특징이나 업무의 특징들을 정확히 이해하고, 그 상황에 맞게 문제의 원인을 확인하고, 문제의 원인을 해결하기 위한 최적의 튜닝 방법을 찾아야 한다는 것이다.

튜닝의 기본 목표는 자원을 상황에 맞게 효율적으로 사용해서 원하는 결과값을 원하는 시간 내에 받아보는 것이다. 병렬 기능을 많이 사용한다고 해서 항상 좋은 결과가 나오는 것은 아니다. 또한 시스템의 자원은 한정되어 있다는 것을 항상 명심해야 한다. 하나의 애플리케이션이 한정된 시스템 자원을 병렬 기능을 사용해서 독점한다면, 다른 애플리케이션들은 상대적으로 피해를 보게 되는 것이다. 즉 튜닝은 하나의 애플리케이션을 위한 것이 아니라 모든 애플리케이션이 조화롭게 운영될 수 있도록 하여야 한다는 것이다.

이런 입장에서 필자는 나무를 보지 말고 숲을 보라고 항상 강조한다. 즉 단위 SQL 문장의 플랜(plan) 튜닝도 중요하지만, 전체적인 조화를 이루는 SQL 문장의 유형 및 구조적인 문제 등이 더 중요하다는 것이다. 업무 시작 이후에 구조적인 문제를 해결하려면 상당한 인적, 시간적 자원을 소모해야 하지만, SQL 문장의 플랜적인 튜닝은 해당 SQL 문장의 튜닝결과를 적용하는 것으로, SQL 문장의 구조적, 유형적 튜닝에 비해 상대적으로 적은 비용이 들 것이다. 그래서 튜닝 작업을 이런 측면에서 바라보도록 항상 노력하여야 할 것이다.

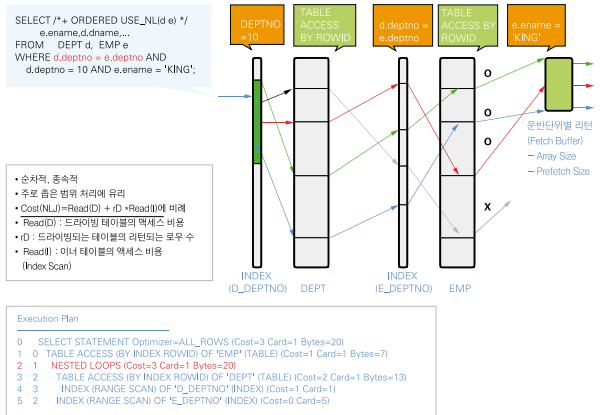
조인 메소드별 특징

오라클의 조인 메소드는 Nested Loop Join(NLJ), Sort Merge Join(SMJ), Hash Join(HJ)의 3가지가 있다. 이들 3가지 조인 메소드로 여러 조인 타입(Basic(natural) Join, Outer Join, Semi Join, Anti Join 등)을 지원하게 된다. Oracle Optimizer는 최적화 단계에서 이들 조인 메소드들 중 조인에 대한 Selectivity와 Cardinality의 계산에 의해 가장 효율적인 것을 선택하게 된다. 단 RBO(Rule Base Optimizer)에서는 인위적인 힌트(USE_HASH)를 주지 않고서는 Hash Join은 전혀 고려하지 않는다. 힌트를 준 것 자체가 이미 RBO가 아니라 CBO(Cost Base Optimizer)로 동작되는 것이다.

조인을 확인할 때 조인 메소드뿐만 아니라 조인 순서 또한 중요하다. 먼저 액세스되는 쪽을 '드라이빙 테이블(Driving Table)' 이라고 하며, 나중에 액세스되는 테이블을 '이너 테이블(Inner Table)' 이라고 한다. Hash Join에서는 'Build Table' 과 'Probing Table' 이라는 용어로 사용된다. SQL 문장에서 튜닝을 잘 하기 위해서는 조인 메소드별 특징을 정확히 이해하고 플랜을 통해 처리되는 과정을 그려볼 수 있어야 한다.

Nested Loop Join(NLJ)

- NLJ는 순차적인 처리로 Fetch의 운반단위(Array Size, Prefetch Size)마다 결과 로우(row)를 리턴 받을 수 있다.
- 첫 번째 로우를 받는 시간은 빠르나, 전체 결과 로우를 받는 데까지 걸리는 시



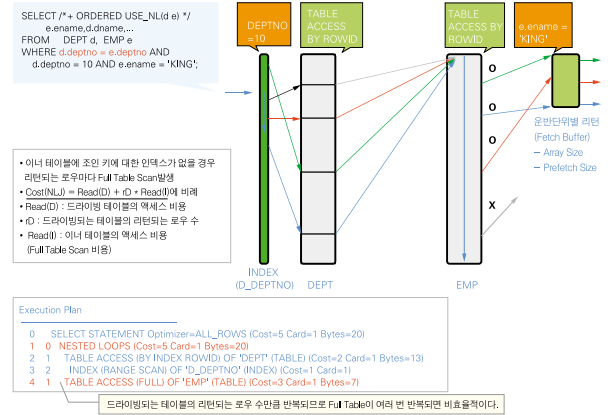
<그림 1> Nested Loop Join - 이너 테이블에 인덱스가 있을 경우

간은 느리다. 즉 첫 번째 로우를 받을 준비가 되어 있는 단계까지를 실행시간으로 볼 때 실행시간은 빠르나 Fetch 시간은 느리다.

- NLJ는 메모리가 필요 없는 조인이다. 그러므로 추가적인 메모리 비용이 필요 없다.
- NLJ는 드라이빙 테이블에서 많은 로우들이 필터링되어 이너 테이블로 찾아 들어가는 부분을 줄여야 하므로 드라이빙 순서가 중요하다.
- 이너 테이블은 드라이빙 테이블의 리턴되는 모든 로우들에 대해서 반복 실행하므로 액세스 효율이 좋아야 한다. 즉 대부분의 경우 이너 테이블은 인덱스가 있어야 한다. 또한 인덱스의 효율이 좋아야 한다. 이너 테이블이 작더라도 액세스 횟수가 많다면 인덱스가 있어야 한다. 인덱스의 효율이 좋지 않아 전체의 Index Range Scan과 같은 경우는 최악의 조건이다.
- NLJ는 주로 인덱스 위주의 심플 블록 I/O의 랜덤 I/O 위주이므로 OLTP에서 적은 데이터 범위 처리에 주로 사용된다. 즉 전체의 15% 이상의 경우는 Full Table Scan을 이용한 Sort Merge 또는 Hash Join을 이용한다.
- NLJ도 드라이빙 테이블이 Full Table Scan에 병렬로 처리되면 이너 테이블도 병렬로 종속적으로 처리된다.

Sort Merge Join(SMJ)

- 전체 로우를 리턴 받는 시간이 빠르다. 즉 첫 번째 로우를 리턴 받을 준비까지의 시간은 느리지만, 준비가 된 상태에서의 Fetch 시간은 빠르다(메모리에서 리턴하므로). 이는 대상 로우들(Where 조건에 의해 필터된 로우들만 정렬)을 가지고 정렬 작업(모든 로우들을 조인 키로 정렬)을 하기 전까지는 어떠한 로우도 리턴할 수 없기 때문이다.
- NLJ와 같이 드라이빙 테이블의 리턴되는 로우 수와 이너 테이블의 액세스 패턴에 의해 액세스 효율이 좌우되지 않으며, 조인 테이블 간에 자신의 처리범위로만 처리량을 결정하므로 독립적이다.
- 추가적인 정렬 메모리(SORT_AREA_SIZE) 비용이 필요하다. 메모리가 부족하면 TEMP 테이블스페이스에 정렬 중간단계(Sort Runs)를 기록하게 되므로 추가적인 디스크 I/O비용이 발생할 수 있다.
- 정렬 메모리에 위치하는 대상은 조인 키뿐만 아니라 Select List도 포함함



<그림 2> Nested Loop Join - 이너 테이블에 인덱스가 없는 경우

로 불필요한 Select List는 제거해야 한다.

- 정렬 작업의 CPU 사용에 대한 오버헤드가 있다. 그러므로 많은 로우들과 전체적으로 Select List의 사이즈의 합이 큰 테이블의 조인에는 문제가 있다. 즉 디스크 정렬을 피할 수가 없으며, 정렬에 CPU 비용이 많이 든다.
- 디스크 정렬만 발생하지 않는다면 넓은 범위 처리에 유리하다.
- 디스크 정렬을 피할 수 없는 경우라면(Batch Job, Create index,...) SORT_AREA_SIZE, SORT_MULTIBLOCK_READ_COUNT를 SQL마다 세션 레벨에 할당해서 사용하도록 한다(WORKAREA_SIZE_POLICY가 Manual일 경우나 Oracle9i/Database 이전 버전에서). 또한 TEMP 테이블 스페이스의 Extent Size도 충분히 크게 주도록 한다.

ALTER SESSION SET SORT_AREA_SIZE= 104857600;

ALTER SESSION SET SORT_AREA_RETAINED_SIZE= 104857600;(같이 준다)

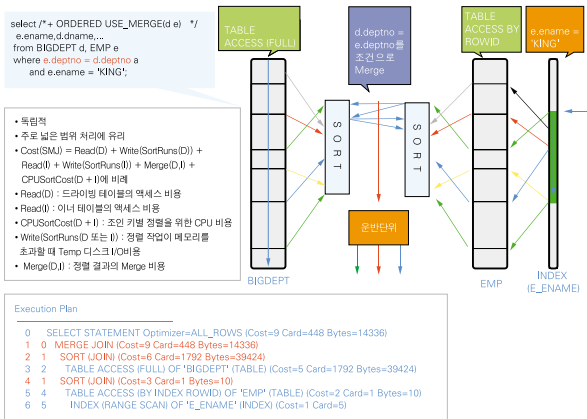
ALTER SESSION SET SORT_MULTIBLOCK_READ_COUNT=128;

- 정렬 메모리의 크기는 (= Target rows × (total selected column's bytes) × 2) 이상 설정하되, PGA의 메모리 한계로 인해 테스트를 통해 PGA Memory Allocation Error가 발생하지 않는 범위 내에서 설정하도록 한다. 필요시 10032 Trace를 이용해 점검한다.

ALTER SESSION SET EVENTS '10032 TRACE NAME CONTEXT FOREVER ;

Hash Join(HJ)

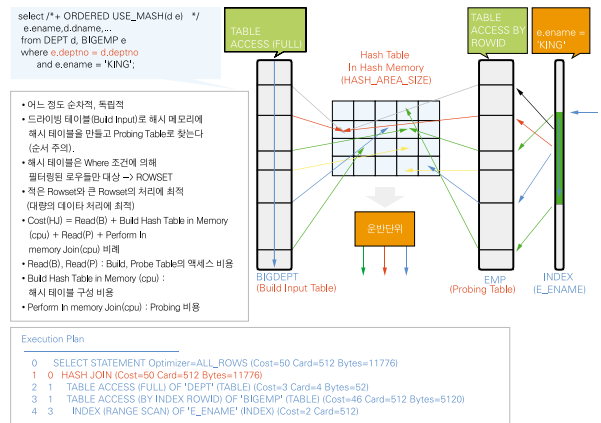
- Hash Join은 두 개의 조인 테이블 중 Small Rowset(Where 조건에 의해 필터링된 로우 수가 작은 테이블)을 가지고 HASH_AREA_SIZE에 지정된 메모리 내에 해시 테이블을 만든다.
- 해시 테이블을 만든 이후부터는 NLJ의 장점인 순차적인 처리 형태이다. 그러므로 NLJ와 SMJ의 장점을 가지고 있다.
- Hash Join은 Basic Join('=')만 가능하다.
- NLJ와 같이 드라이빙 테이블의 리턴되는 로우 수와 이너 테이블의 액세스 패턴



<그림 3> Sort Merge Join

탄에 의해 액세스의 효율이 좌우되지 않으며, 조인 테이블 간에 자신의 처리 범위로만 처리량을 결정하므로 독립적이다.

- SMJ의 단점인 많은 로우들의 처리 또는 전체적으로 Select List의 사이즈의 합이 큰 테이블의 조인시 정렬 작업의 CPU 사용에 대한 오버헤드 및 디스크 정렬과 같은 문제점은 없다. 그러므로 최소한 SMJ보다는 우수하다.
- 한 테이블은 작은 Rowset 사이즈(리턴되는 로우 수와 Select List 기준), 다른 테이블은 아주 큰 사이즈의 조인에 유리하다. 이러한 경우는 반드시 작은 사이즈를 가지고 해서 테이블을 만들어야 한다. 단, Hash Join은 순서가 매우 중요하다는 점에 주의하는데, 작은 Rowset으로 해서 테이블을 만들어야 하기 때문이다.
- 힌트를 잘못 주어서 Big Rowset이 리턴되는 테이블부터 드라이브된다면 (Build Table), HASH_AREA_SIZE의 메모리 부족으로 TEMP 디스크 I/O가 발생한다. 그러므로 힌트를 줄 경우 반드시 드라이브 순서를 정확히 주어



<그림 4> Hash Join

야 한다.

- 디스크 I/O를 피할 수 없는 경우라면, HASH_AREA_SIZE(default : =SORT_AREA_SIZE × 2)를 SQL마다 세션 레벨에 할당해서 사용하도록 한다(WORKAREA_SIZE_POLICY가 Manual일 경우이거나 Oracle9i Database 이전 버전에서). 또한 TEMP 테이블스페이스의 Extent Size도 충분히 크게 주도록 한다. HASH_MULTIBLOCK_IO_COUNT는 옵티마이저에게 자동 조정하도록 설정하지 않는다.

ALTER SESSION SET HASH_AREA_SIZE=104857600;

- 해서 메모리의 사이즈는 (= Small Table의 Target rows × (total selected column's bytes) × 1.5) 이상 설정하되, PGA의 메모리의 한계로 인해 테스트를 통해 PGA Memory Allocation Error가 발생하지 않는 범위 내에서 설

Key	Hash1	Partition # (2 prattions)	Hash2	Bitvec Pos (32 bits)	Bucket # (8 bcks)
ANDY	267B4DCD	1	807F5A3E	1E	6
POONAM	7AF8C9A7	1	3E5CEEDC	1C	4
DEEPAK	EE5FFFFE	0	D61F8268	08	0
RUSS	4403A5DA	0	AB31D505	05	5
SUNITHA	32C9A7C6	0	C9F92A85	05	5
RAMESH	3A754F8D	1	E9000EFB	1B	3
JONATHAN	DE3525B1	1	10C2B4BC	1C	4
RICHARD	DF970C35	1	16A2D766	06	6

Assuming,
Partition# mask = 0x00000001
Bitvec Position mask = 0x0000001F
Hash Bucket mask = 0x00000007

hash buckets (kkrhht)

row headers (kkrhlt)

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

row data

Partition 0

bit vector filter (kkrhpbtt)

00000100 00000000 00000000 00000000

row data

RUSS

Partition 1

bit vector filter (kkrhpbtt)

00000010 00000000 00000000 00001010

row data

ANDY

JONATHAN

RICHARD

English Table

Name

ANDY

RUSS

JONATHAN

RICHARD

BugEsc Table

Name

ANDY

POONAM

DEEPAK

RUSS

SUNITHA

RAMESH

JONATHAN

Select /*+ use_hash(BugEsc) */
English.Name from English, BugEsc
Where English.Name = BugEsc.Name;

1. Build Table단계 : Smaller Table(English)를 스캔하여 메모리상에 해서 테이블을 만든다.
해서 테이블은 Partitioned, Bit Vectors, Hash Bucket으로 구성되며, 해서 테이블을 만드는
과정에서 해서 메모리가 부족할 경우 TEMP 디스크로 파티션의 일부 또는 전부를 보관한 뒤
2번째 단계에서 처리하게 된다.

	Partition#	Bit Vector
ANDY	1	1E
RUSS	0	05
JONATHAN	1	1C
RICHARD	1	06

2. Probing Table단계 : 해서 테이블을 만든 이후, Larger Table(BugEsc)를 스캔하여
조인 키에 대한 Hash Function #1, #2를 적용하여 해당 파티션 및 버킷에 같은 조인 키가
있는지 확인하여 있으면, 즉 조인이 성립되면 로우들을 리턴하고 없으면 로우를 버리게 된다.

ANDY Passes bitvec. Found on chain 6 **Row returned.**
POONAM Partition 1 and bitvec (1C) is set. However,
row not found on chain 4.
DEEPAK Partition 0 but bitvec (08) unset.
RUSS Passes bitvec. chain 5. **Row returned.**
SUNITHA Partition 0, bitvec bitvec 5 set,
however, not found on chain 5.
RAMESH Partition 1 but bitvec (1B) unset.
JONATHAN partition 1 on chain 4. **Row returned.**

<그림 5> Hash Join Detail

정하도록 한다. 필요시 10104 Trace를 이용해 점검한다.

```
ALTER SESSION SET EVENTS '10104 TRACE NAME CONTEXT FOREVER ;
```

SQL 튜닝시 고려사항

SQL 튜닝시에 고려해야 할 점을 정리하면, 다음과 같다.

- 가능한 힌트는 사용하지 않는다. 힌트를 많이 구사한 애플리케이션들은 지난 호에서 설명했던 것과 같이 기준(파라미터 및 통계정보)이 잘못 설정된 경우가 많다. 힌트는 최후에 어쩔 수 없는 경우에 사용하도록 한다.
 - 1차적으로 플랜이 원하는 경우가 아닐 경우 통계정보 및 Init.ora의 파라미터 값들을 확인해 본다.
[USER|ALL|DBA]_TABLES, [USER|ALL|DBA]_INDEXES, [USER|ALL|DBA]_TAB_COLUMNS 등의 디셔너리 정보 확인, 최종 분석 시간, 블록 수, 로우 수, 칼럼의 Distinct 값, 인덱스의 Clustering Factor, Sample Size 등을 확인한다. 이들 값들이 현실 데이터와 비슷한지 확인한다. 통계정보가 없거나 너무 오래됐거나 샘플링 사이즈가 너무 작은 경우, 현실 데이터와 다를 수 있다. 이러한 경우는 통계정보를 다시 생성한다.
 - 칼럼에 대한 통계정보(히스토그램)는 안 돌리는 것을 원칙으로 한다. 그러나 편향된 데이터 분포도를 가지고 있다면 히스토그램을 운영한다. 또한 이들 칼럼에 대한 Where 절의 사용되는 값들은 바인드 변수를 사용하지 않도록 한다. 히스토그램을 사용하는 곳에는 상수(literal) 값을 사용하여야 플랜이 효과적으로 풀린다.
 - 힌트를 지정할 경우는 가능한 타이트하게 주도록 한다. 그렇지 않을 경우 향후 플랜이 변경될 가능성이 많기 때문이다.
- 예> /*+ USE_NL(a b) */ => /*+ ORDERED USE_NL(a b) */
- 힌트는 힌트의 의미를 정확히 이해하고 합당한 힌트를 주도록 한다. 어설픈 힌트는 오히려 역효과가 발생하는 경우가 많다.
 - 통계정보는 운영중에 직접 돌리지 않는다.
 - 집중적인 운영시기에 통계정보 수집을 위한 실행은 Library Cache Contention을 유발하며, 관련 SQL 및 PL/SQL들을 Invalid시켜 성능저하 및 문제의 원인이 되기도 한다.
 - 저녁시간의 한가한 시간을 이용해서 돌린다(특히 시스템의 집중 사용시기 등에 유의).
 - WORKAREA_SIZE_POLICY=AUTO이면 *_AREA_SIZE는 이용하지 않으며, 설정해봐야 의미가 없다. 즉 옵티마이저는 *_AREA_SIZE에 의해 플랜을 결정하지 않는다.

- 튜닝시 플랜적인 튜닝뿐만 아니라 구조적인 튜닝에도 집중한다.

- Execution이 높은 것은 과다한 Loop Query가 아닌지 검토한다. Loop Query의 보완, 최적화가 필요하다.
- 일회성(상수를 사용) 비공유 SQL, 특히 집중적으로 실행되는 Literal SQL은 바인드 변수 기법을 사용한다.
- 파싱(parsing)을 줄이는 방법으로 Java의 'Statement Cache', PRO*C의 RELEASE_CURSOR=NO 등 프로그래밍 언어의 효과적인 기법들을 사용한다.
- Array Processing 기능을 사용한다. 기본적으로 거의 대부분의 DB 접속 방식에서 이들 기법들을 제공하고 있으며, 코딩상의 특별한 처리 없이 PREFETCH 기능이 이와 같은 기능이며, 애플리케이션 개발시 특별한 구현 작업 없이 옵션 설정만으로 가능하다(ODBC, JDBC, OO4O, ADO, PRO*C 등).
- PL/SQL의 Bulk Binding/Bulk Collecting 기능을 이용한다.
- Aggregate Function 등 향상되고 효과적인 여러 질의 기능을 활용한다.

- 튜닝시 플랜은 상수로 테스트하지만, 실제로 바인드 변수로 운영되는 경우 플랜이 다를 수 있다. 프로그램에 바인드 변수로 되어 있다면 바인드 변수로 플랜을 확인해 봐야 한다.

- 기타 다음 사항도 고려한다.

- Hash Join을 사용할 경우 드라이빙 순서, Rowset을 고려하여 사용한다.
- SQL 문장에서 반드시 필요한 칼럼만 선택한다. 불필요한 칼럼들은 정렬, 해시 작업에서 메모리에 로딩해야 하므로 TEMP 디스크 I/O의 원인이 되기도 한다.
- Chaining % 비율을 항상 검토하고 Row Chaining 비율이 높은 테이블에 대해서는 칼럼의 데이터 타입 조정 및 블록의 PCTFREE 등을 늘리도록 한다. 테이블의 구조적인 문제 또는 업무적인 형태를 고려하여 Reorg를 하도록 한다(CTAS, MOVE, Exp/Imp 등 이용).

- 평균 로우 길이와 블록당 로우 수도 항상 주의 깊게 관찰하여 문제점이 없는지 검토한다(통계정보 이용 DBA_TABLES.NUM_ROWS/DBA_TABLES.BLOCKS). 블록당 로우가 적은 경우는 DELETE가 많이 된 경우이므로, Full Table Scan이 자주 발생된다면 Reorg 대상이 될 수 있다. 그러나 RAC 환경에서는 블록 경합을 줄이기 위해 인위적으로 PCTFREE를 키워 블록당 로우 수를 적게 가져가는 경우도 있다.

- Hash Join과 Sort Merge Join시 TEMP쪽에 I/O가 발생하지 않도록 한다.

- PRO*C 애플리케이션일 경우, 바인드 변수의 사용여부, RELEASE_CURSOR=NO, PREFETCH=1000(batch), PREFETCH=100(OLTP)를 권장한다.

Execution Plan 보기

Execution Plan이란 옵티마이저가 질의 최적화 단계에서 RBO 또는 CBO에 의해 결정해낸 최적의 액세스 경로 정보를 가지고 QEP Generator가 만들어낸 실행계획이다. 이 Execution Plan은 SQL 문장이 실행될 때 필요한 모든 정보를 포함하고 있다.

- 액세스 경로 : 어떠한 방법으로 데이터에 접근할 것인가? (Index Scan, Index Fast Full Scan, Full Table Scan 등)
- 조인 메소드 : 어떤 조인 메소드를 사용할 것인가?
- 조인 순서 : 어떠한 조인의 순서로 풀릴 것인가?

다음과 같은 Execution Plan을 가정하자.

ID PID Execution Plan

```
-----
0  SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=5 Bytes=250)
1  0  TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=1 Card=5 Bytes=160)
2  1  NESTED LOOPS (Cost=3 Card=5 Bytes=250)
3  2  TABLE ACCESS (BY INDEX ROWID) OF 'DEPT' (TABLE) (Cost=2 Card=1 Bytes=18)
4  3  INDEX (RANGE SCAN) OF 'DEPT_DEPTNO' (INDEX) (Cost=1 Card=1)
5  2  INDEX (RANGE SCAN) OF 'EMP_DEPTNO' (INDEX) (Cost=0 Card=5)
```

플랜에서 나오는 각 라인을 '로우 소스(Row Source)'라고 한다. 플랜을 보면서 처리 순서를 판단하는 것은 간단하다. 플랜은 트리 형태로 되어 있으며, 자신보다 하위 레벨이 있으면 하위 레벨부터, 같은 레벨이라면 위(상)의 로우 소스부터 실행된다.

위 플랜의 'Optimizer=CHOOSE'에서 알 수 있듯이 해당 SQL 문장은 옵티마이저 모드가 CHOOSE에서 플랜이 만들어진 것이다. 또한 플랜에서 'Cost='의 항목이 나오면 CBO로 풀렸다는 것이다. RBO인지 CBO인지의 판단은 옵티마이저 모드의 항목으로 판단하는 것이 아니라 'Cost='로 판단한다는 것에 주의하자.

위 플랜에서 2개의 테이블 DEPT와 EMP 테이블 각각의 액세스 경로를 확인할 수 있다. 모두 인덱스를 사용하고 있는 것이다. 또한 조인 메소드로는 Nested Loop Join이 사용되었다. 조인 순서는 자신보다 하위 레벨이 있으면 하위 레벨부터, 같은 레벨이라면 위의 로우 소스부터 실행된다는 법칙을 적용해 보면, ID를 기준으로 4->3->5->2->1->0의 순서로 처리된다. 단 조인 메소드가 Nested Loop이기 때문에 3에서 리턴되는 로우 수만큼 다음 단계가 반복된다. 그러므로 조인 순서는 DEPT -> EMP 순으로 Nested Loop로 처리될 것이라는 것을 알 수 있다.

또한 'Card=5'는 Computed Cardinality를 나타내며, 몇 건의 로우가 리턴될 것인지를 CBO가 통계정보를 이용해서 계산해낸 값이다. 'Bytes=250'은 리턴될 로우들의 바이트를 나타내므로 5 로우에 250바이트

정도의 리턴 로우가 발생할 것이라는 것을 예측할 수 있다. 통계정보만 정확하다면 이들 값도 상당히 정확하다고 보면 된다.

Execution Plan을 제대로 보기 위해서는,

- 각 데이터베이스 사용자마다 PLAN_TABLE이 있어야 하는데, PLAN_TABLE은 오라클 버전마다 다르다. 해당 오라클 버전의 \$ORACLE_HOME/rdbms/admin/utlxplan.sql을 실행하면 만들어진다.
- SQL을 실행하지 않고 Trace만 보는 방법도 있다.
- 'EXPLAIN PLAN' <리스트 1>, SQL*Plus의 'SET AUTOTRACE TRACEONLY EXPLAIN' <리스트 2>.
- 플랜을 PLAN_TABLE에서 확인할 수도 있다.

Oracle8i Database 이전까진 Plan_Table에서 직접 선택하고, Oracle8i Database 이상부터는 Plan_Table에서 직접 선택하거나 \$ORACLE_HOME/rdbms/admin 위치에서 utxppls.sql(Serial Plan) 또는 utlxplp.sql(Parallel Plan) 스크립트를 실행하면 된다. Oracle9i Database 이전에서는 utxppls.sql, utlxplp.sql 외에 'select * from table(dbms_xplan.display);'

<리스트 1> Execution Plan 보기(EXPLAIN PLAN) 실행 예
(Oracle9i Database, Oracle10g Database)

-- Oracle9i Database R2 Sample (9.2.0.4)

SQL> SET LINESIZE 130

SQL> SET PAGESIZE 0

SQL>

SQL> EXPLAIN PLAN

2 SET STATEMENT_ID = 'TEST_MYSQL'

3 FOR SELECT ename, job, sal, dname

4 FROM emp, dept

5 WHERE emp.deptno = dept.deptno

6 AND NOT EXISTS

7 (SELECT *

8 FROM salgrade

9 WHERE emp.sal BETWEEN losal AND hisal);

EXPLAIN PLAN...FOR
<SQL>로 플랜을
작성한다.

Explained.

SQL>

SQL> -- 오른쪽의 Script로 실행해도 됨. @?/rdbms/admin/utlxpls.sql

SQL> select * from table(dbms_xplan.display);

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		14	476	12
1	MERGE JOIN ANTI		14	476	12
2	SORT JOIN		14	392	8
*3	HASH JOIN		14	392	5
4	TABLE ACCESS FULL	EMP	14	238	2
5	TABLE ACCESS FULL	DEPT	4	44	2
*6	FILTER				
*7	SORT JOIN				
8	TABLE ACCESS FULL	SALGRADE	5	30	2

단위 스텝별 예상
로우 및 바이트를
예측할 수 있다.
상위 단계 하위 단
을 포함한다.

Predicate Information (identified by operation id):

3 - access("EMP"."DEPTNO" = "DEPT"."DEPTNO")
6 - filter("EMP"."SAL" <= "SALGRADE"."HISAL")
7 - access("EMP"."SAL" >= "SALGRADE"."LOSAL")
filter("EMP"."SAL" >= "SALGRADE"."LOSAL")

Note: cpu costing is off

24 rows selected.

-- 10g Sample (10.1.0.2)

SQL> -- @?/rdbms/admin/utlxpls.sql

SQL> select * from table(dbms_xplan.display);

Oracle9i Database부터
Access & Filter Predicate
정보가 추가되었다.
이 정보를 이용해서 필터 조건 및
조인 조건을 확인할 수 있다.

Oracle10g Database부터 다폴
트로 CPU 반영 비중과 예측시간
을 확인할 수 있다.
Oracle10g Database부터는 비
용 기준이 시간이므로 가능하다.

Id	Operation	Name	Rows	Bytes	Cost	%CPU	Time
0	SELECT STATEMENT		14	532	18	(17)	00:00:01
1	MERGE JOIN ANTI		14	532	18	(17)	00:00:01
2	SORT JOIN		14	420	12	(17)	00:00:01
*3	HASH JOIN		14	420	11	(10)	00:00:01
4	TABLE ACCESS FULL	DEPT	4	52	5	(0)	00:00:01
5	TABLE ACCESS FULL	EMP	14	238	5	(0)	00:00:01
*6	FILTER						
*7	SORT JOIN		5	40	6	(17)	00:00:01
8	TABLE ACCESS FULL	SALGRADE	5	40	5	(0)	00:00:01

Predicate Information (identified by operation id):

3 - access("EMP"."DEPTNO" = "DEPT"."DEPTNO")
6 - filter("EMP"."SAL" <= "HISAL")
7 - access("EMP"."SAL" >= "LOSAL")
filter("EMP"."SAL" >= "LOSAL")

<리스트 2> Execution Plan 보기(SQL*Plus의 SET AUTOTRACE) 실행 예
(Oracle9i Database)

9R2 Sample (9.2.0.4) =====>

SQL> SET AUTOTRACE ON

SQL>

SQL> SELECT ename, job, sal, dname

2 FROM emp, dept

3 WHERE emp.deptno = dept.deptno

4 AND NOT EXISTS

5 (SELECT *

6 FROM salgrade

7 WHERE emp.sal BETWEEN losal and hisal);

no rows selected

(1) SQL 문장이 실행되었다. 실행은 하지 않고 플랜만 확인하려면
'SET AUTOTRACE ONLY EXPLAIN' 으로 처리한다.

Execution Plan

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=14 Bytes=756)
1 0 MERGE JOIN (ANTI) (Cost=13 Card=14 Bytes=756)
2 1 SORT (JOIN) (Cost=8 Card=14 Bytes=392)
3 2 HASH JOIN (Cost=5 Card=14 Bytes=392)
4 3 TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=238)
5 3 TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=44)
6 1 FILTER
7 6 SORT (JOIN)
8 7 TABLE ACCESS (FULL) OF 'SALGRADE' (Cost=2 Card=409 Bytes=10634)

(2) 내부적으로 EXPLAIN PLAN...으로 처리된 플랜 정보를 보여준다.
실제 Runtime Plan은 아니다.

Statistics

```

0 recursive calls
0 db block gets
21 consistent gets
0 physical reads
0 redo size
376 bytes sent via SQL*Net to client
372 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
2 sorts (memory)
0 sorts (disk)
0 rows processed

```

(3) 실제 실행된 Execution Statistics를 보여준다. 전체 읽어들이 블록 수는 db block gets + consistent gets.

Cached Execution Plan(V\$SQL_PLAN)

Explain Plan으로 보는 플랜과 실제 실행시 플랜이 다를 수 있다. 이는 Explain Plan은 단지 SQL 문장에 대한 구조적인 분석에 예상 플랜을 만들어내기 때문이다. 예를 들어, 'select * from emp where empno = :B1'의 SQL 문장을 실행한다고 생각해 보자.

Empno가 인덱스가 설정되어 있고 Character 타입으로 되어 있으며, 인덱스를 이용하는 것이 효과적이라고 가정하면 Explain plan은 Empno의 인덱스를 이용해서 풀릴 것이다. 그러나 실제 실행시 바인드 변수인 'B1'에 Character 타입이 아니라 Number 타입으로 바인드되었다면 인덱스를 이용할 수 없는 것이다. 또한 Oracle9i Database의 Bind Peeking과 같은 기능은 처음 바인딩되는 상수에 의해 플랜을 결정하는데, 이 방식에 의해서도 그러한 상황이 있을 수 있다.

이와 같이 Explain Plan을 보는 것과 실제 예측 실행시간이 너무 차이가 난다면 Runtime Plan을 확인해 볼 필요가 있다. Oracle9i Database부터 V\$SQL_PLAN의 성능 뷰를 제공하며, 현재 캐쉬화되어 있는 SQL 문장들에 대한 Runtime Plan을 확인해 볼 수 있다. V\$SQL_PLAN은 PLAN_TABLE과 칼럼 항목이 거의 같다.

```

SELECT hash_value, (select sql_text from v$sql s where s.hash_value = p.hash_value and
s.address = p.address and rownum <= 1), child_number, ID, PARENT_ID, LPAD(
'2*(depth)||OPERATION||DECODE(OTHER_TAG,NULL,'','*')|| DECODE(OPTIONS,NULL,'','
'||OPTIONS||')'||DECODE(OBJECT_NAME,NULL,'',' OF '||OBJECT_NAME||'')||
DECODE(OBJECT#,NULL,'','(Obj# ||TO_CHAR(OBJECT#)||')'||DECODE(ID,0,DECODE(OPTIMI-
ZER,NULL,'',' Optimizer='||OPTIMIZER)||DECODE(COST,NULL,'','
(Cost='||COST||DECODE(CARDINALITY,NULL,'',' Card='||CARDINALITY)||DECODE(BYTES,NULL,'','
Bytes='||BYTES)||') ) SQLPLAN,OBJECT_NODE,PARTITION_START,PARTITION_STOP,
PARTITION_ID,CPU_COST,IO_COST,TEMP_SPACE,DISTRIBUTION,OTHER,

```

```

ACCESS_PREDICATES, FILTER_PREDICATES FROM v$sql_plan p
START WITH ID=0 and hash_value = [XXXXXXXXXX]
CONNECT BY PRIOR ID=PARENT_ID AND
PRIOR hash_value=hash_value AND
PRIOR child_number=child_number
ORDER BY hash_value,child_number,ID,POSITION

```

SQL_TRACE와 TKPROF를 이용한 SQL 튜닝

SQL_TRACE 또는 10046 Trace Enable/Disable

SQL_TRACE는 애플리케이션이 SQL 문장들을 처리하는 과정을 Trace로 남기게 하는 기능이다. 10046 Trace 기능은 SQL_TRACE의 기능에 추가적인 정보를 기록한다. 레벨 1은 SQL_TRACE 기능과 같으며, 레벨 4는 바인드 변수 정보, 레벨 8은 Wait Event 정보, 레벨 12는 바인드 변수 정보와 Wait Event 정보를 같이 보여준다.

주의할 점은 Trace를 On 했으면 반드시 모니터링 후 Off 해야 한다는 것이다. 그렇지 않을 경우 Disk Full이 발생할 수 있으므로 반드시 주의해야 한다. Trace는 init.ora의 user_dump_dest에서 지정된 곳에 생성된다.

- 인스턴스 레벨 : init.ora 파라미터 이용

```
sql_trace = {TRUE | FALSE}
```

또는

```
event = "10046 trace name context forever, level {1|4|8|12}"
```

- 세션 레벨 : SQL *Plus 또는 애플리케이션 루틴 내

```
ALTER SESSION SET SQL_TRACE = {True|False};
```

또는 10046 Trace On

```
alter session set events '10046 trace name context forever, level {1|4|8|12}';
```

10046 Trace Off

```
alter session set events '10046 trace name context off';
```

또는

```
EXECUTE dbms_session.set_sql_trace({True|False});
```

또는

```
EXECUTE dbms_system.set_sql_trace_in_session(session_id, serial_id, {True|False});
```

Execution Plan 보기(SQL_TRACE, 10046 Trace와 TKPROF) 실행 예 (Oracle9i Database)

Oracle9i Database Release 2(9.2.x)부터는 튜닝 대상이 어느 곳인지(플랜 상의 스텝) 판단하기 쉽게 획기적으로 개선되어, 초보자도 튜닝 대상을 쉽게 찾을 수 있다. 또한 사용자가 SQL 문장을 실행해서 결과 값을 받는 서비스

타임은 'DB 실행시간 + 대기시간' 이다. Oracle9i Database부터는 SQL TRACE의 레벨에 따라 Wait 정보의 요약 정보도 같이 보여주므로 어느 곳에 병목현상이 있는지 판단하기 쉬워졌다. 다음은 Oracle9i Database Release 2(9.2.x)부터 개선된 사항이다<리스트 3>.

- 10046 Trace 레벨에 따라 Wait(레벨 8, 레벨 12일 경우) 정보도 표시
- 각 로우 소스(플랜상의 스텝)마다 Statistics 표시
- Oracle9i Database에서는 time=xxxxxxx 정보가 1/1000000초 단위로 나타난다. Oracle8i Database까지는 1/100초였다.
- Runtime Plan & TKPROF 실행시 플랜 주의
- TKPROF ... EXPLAIN=xxxx/yyyy일 경우 플랜이 2개(Runtime Plan & Tkprof 실행 시점의 플랜)

<리스트 3> Oracle9i Database R2(9.2.0.4) 샘플

```
-- alter session set sql_trace=true;
alter session set events '10046 trace name context forever, level 12';
```

```
SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND NOT EXISTS
(SELECT *
FROM salgrade
WHERE emp.sal BETWEEN losal AND hisal);
```

```
ORA9iR2L@oracle> tkprof ora9i2l_ora_2137.trc ora9i2l_ora_2137.prf
explain=scott/tiger width=132
```

TKPROF: Release 9.2.0.4.0 - Production on Mon Nov 22 16:30:54 2004

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

```
ORA9iR2L@oracle> vi ora9i2l_ora_2137.prf
```

```
SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
```

AND NOT EXISTS

```
(SELECT *
FROM salgrade
WHERE emp.sal BETWEEN losal AND hisal)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
(1) Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.06	0.09	0	21	0	0
total	3	0.06	0.09	0	21	0	0

실행 수 Sec 처리된 처리된
블록 수 로우 수

Misses in library cache during parse: 0

```
(2) Optimizer goal: CHOOSE
Parsing user id: 59 (SCOTT)
```

Rows	Row Source Operation
0	MERGE JOIN ANTI (cr=21 r=0 w=0 time=94397 us)
14	SORT JOIN (cr=14 r=0 w=0 time=92823 us)
(3) 14	HASH JOIN (cr=14 r=0 w=0 time=92237 us)
14	TABLE ACCESS FULL OBJ#(30627) (cr=7 r=0 w=0 time=860 us)
4	TABLE ACCESS FULL OBJ#(30628) (cr=7 r=0 w=0 time=275 us)
14	FILTER (cr=7 r=0 w=0 time=1238 us)
40	SORT JOIN (cr=7 r=0 w=0 time=856 us)
5	TABLE ACCESS FULL OBJ#(30630) (cr=7 r=0 w=0 time=383 us)

Oracle9i Database Release 2부터는 각 로우 소스(스텝)별 일량을 확인할 수 있는 통계정보를 보여준다.

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
0	MERGE JOIN (ANTI)
14	SORT (JOIN)
(4) 14	HASH JOIN
14	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'EMP'
4	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'DEPT'
14	FILTER
40	SORT (JOIN)
5	TABLE ACCESS (FULL) OF 'SALGRADE'

Elapsed times include waiting on following events:

Event waited on	Times	Max. Wait	Total Wait
(5) ----- Waited -----			
SQL*Net message to client	1	0.00	0.00
SQL*Net message from client	1	1.57	1.57

(1) SQL 문장 처리 정보

- query : Consistent Read(CR)
current : Current Read(SCUR)
disk : Physical Read
- Parse : 파싱(Parse Request 수, 실제 하드 파싱은 아님)
Execute : SQL 문장 실행 수
Fetch : Fetch 횟수
- cpu, elapsed : CPU 사용시간 및 전체 처리시간
- Logical Read = Query + Current(Logical Read는 Physical Read를 포함한다. 그러므로 Logical Read >= Physical Read이다. 그러나 반대의 경우라면, 임시 테이블스페이스로 디스크 I/O가 발생했다고 보면 된다(Sort, Hash, Bitmap Operation 등에 의해). 여기서의 관점은 Logical Read는 전체 Block Buffer Operation의 양을 나타내므로 이들 처리 블록 수를 줄이는 것이 튜닝의 관건이다.
- Fetch 횟수와 로우 수와의 관계에서 Fetch = Rows일 경우는 Single Row Fetch의 처리를 하였고, Fetch <= Rows일 경우는 Array Fetch 또는 Prefetch 처리를 한 경우이다. 여기서의 관점은 성능 향상을 위해 Array Fetch 또는 Prefetch를 효과적으로 사용하고 있는지의 관점이다.
- Parse 수와 Execute 관점에서 본다면, 같은 SQL 문장을 매번 실행할 때마다 Parse Request를 할 필요가 없다는 것이다. 이를 줄이는 방법으로 Pro*C의 'RELEASE_CURSOR=NO' (Dynamic SQL은 해당 안됨), Java의 'Statement Cache' 등이 있다.

(2) SQL 문장의 파싱 정보

- 'Misses in library cache during parse'의 값이 0이면 소프트 파싱, 1이면 SGA Cache에 없어서 하드 파싱이 발생한 경우이다.
- 옵티마이저 모드 정보 및 파싱 스키마 정보를 확인할 수 있다.
- Recursive SQL일 경우 Recursive Depth 정보도 나타난다.

(3), (4) SQL 문장 Execution Plan 정보

(Runtime Plan/TKPROF 실행시 플랜)

- TKPROF의 explain=xxx/xxx를 주게 되면 2개의 플랜이 만들어진다. 처음 것이 Runtime Plan이며, 두 번째가 TKPROF 유틸리티를 돌린 시점의 EXPLAIN PLAN에 의해 만들어진 플랜이다. 이들 플랜은 TKPROF를 실행한 시점이 다를 수 있으므로 다를 수가 있는 것이다.
- Rows : Oracle Database 8.0까지는 액세스했던, 즉 찾아 들어갔던 로우 수로 나타나며, Oracle8i Database부터는 조건에 의해 필터되어 리턴된 로우

수를 나타낸다.

- Oracle9i Database Release 2부터는 각 로우 소스(스텝)별 일량을 확인할 수 있는 Statistics정보를 보여준다. 상위 스텝은 하위 스텝의 값들을 포함한다. 그러므로 전체 일량 중에서 병목현상이 어느 곳에서 가장 많이 발생하는지 명확히 알 수 있다. 즉 튜닝의 포인트를 정확히 확인할 수 있다.
- cr = : Consistent Read 총 블록 수
r = (Oracle Database 10g는 pr =) : Physical Read 총 블록 수
w = (Oracle Database 10g는 pw =) : Physical Write 총 블록 수
time = : 전체 처리시간(마이크로초(1/1000000초) 단위)
- 여기서 가장 상위의 cr=21은 (1)의 SQL 문장 처리정보에서 확인한 Logical Read(= Query + Current)의 값과 같다는 것을 확인할 수 있다. 또한 전체 처리시간은 'time=94397' 이므로 약 0.09초 걸렸다는 것을 알 수 있다. 전체 처리 일량 중 병목을 찾고자 한다면 전체 일량 중 가장 많은 블록 처리 및 시간으로 탐다운 식으로 찾아내려가면 누구든지 쉽게 찾을 수 있을 것이다. 해당 부분을 찾은 뒤, 조인 메소드의 변경 -> 조인 순서의 변경 -> 액세스 경로 변경의 중심으로 튜닝을 실시하면 될 것이다.

(5) SQL 문장 Wait 정보

- 이 부분은 10046 Event의 8 또는 12 레벨에 의해 생성되며, SQL 문장을 실행하는 데 발생되었던 Wait 정보의 요약 값을 나타낸다. 사용자가 느끼는 실행시간은 SQL 문장의 실행시간과 Wait 시간의 합이므로 이들 Wait 시간을 주의 깊게 관찰할 필요가 있다.
- Times Waited : Wait Event가 발생했던 횟수
- Max. Wait : 최대로 길었던 Wait 시간(초 단위)
- Total Waited : 전체 Wait 시간(초 단위)

TKPROF 아웃풋의 로우 값의 버전별 변화

TKPROF의 아웃풋 형태가 Oracle8i Database 이전까지는 테이블 또는 인덱스를 찾아들여간 로우 수를 나타내며, Oracle8i Database 서부터 로우 값은 필터링되어 리턴된 로우 수를 나타낸다. TKPROF 아웃풋에 로우 값이 0으로 나오는 경우는 해당 SQL 문장의 커서가 종료되기 전에 Trace가 종료되었거나 끊긴 경우이다. Runtime Plan은 커서가 종료되는 시점에 기록되므로 이 값이 0으로 나올 수 있다.

```
-- Oracle Database 8.0.x의 플랜 예
select /*+ ORDERED USE_NL(d e) */ *
from dept d, emp e
where e.deptno = d.deptno and d.deptno = 10
```

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
3	NESTED LOOPS
4	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'DEPT'
14	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'EMP'

DEPT 테이블에서 Full Table Scan에 의해 4로우가 액세스되었으며, EMP 테이블도 Full Table Scan에 의해 14로우가 액세스되었다. Nested Loop Join에 의해 3로우가 리턴되었다.

```
-- Oracle 9.2.x의 Plan 예
select /*+ ORDERED USE_NL(de) */ *
from dept d, emp e
where e.deptno = d.deptno and d.deptno = 10
```

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
3	NESTED LOOPS
1	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'DEPT'
3	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'EMP'

DEPT 테이블에서 Full Table Scan에 의해 1로우가 리턴되었으며, EMP 테이블도 Full Table Scan에 의해 3로우가 리턴되었다. Nested Loop Join에 의해 3로우가 리턴되었다.

Oracle Database 10g의 튜닝 관련 신기능

Oracle Database 10g에서 튜닝 기능은 더욱 개선, 강화되었다.

• 자동 성능 진단과 튜닝

Oracle Database 10g부터는 자가관리(self-management) 기능이 강화되었으며, Automatic Statistics Collection, Automatic Database Diagnostic Monitoring(ADDM), Automatic SQL 튜닝과 같은 기능을 튜닝에 이용할 수 있다. Automatic Workload Repository(AWR)는 셀프튜닝과 문제진단을 목적으로 성능 정보를 수집하고 처리하고 유지한다. Automatic Database Diagnostic Monitor(ADDM)는 AWR에 수집된 정보를 주기적으로 분석하고 오라클 시스템에 대한 문제의 원인을 진단하고 권장사항을 제공한다.

Oracle SQL Tuning Advisor는 SQL 문장의 최적화를 위해 빠르고 효과적인 방법을 제공한다. 내부적으로 DBMS_SQLTUNE 패키지를 제공하며, 사람이 직접 하던 튜닝을 이제는 Oracle SQL Tuning Advisor의 기능을 이용해 문제의 원인을 상세하게 찾아내고 개선안을 얻을 수 있으며, 개선안을 수용하면 같은 SQL 문장이 실행되었을 경우 튜닝된 결과로 플랜이 처리된다.

• Application End to End Tracing

Application End to End Tracing 기능은 클라이언트 식별자(Login ID), 서비스명(Application Group), 모듈명 또는 액션명으로 시스템 자원을 과다하게 사용하는 SQL 문장과 같은 워크로드의 원인을 찾을 수 있도록 해주는 유용한 기능이다. 즉 멀티터 환경에서의 성능 디버깅을 쉽게 할 수 있는 기능이다.

• trcsess 유틸리티

trcsess는 커맨드라인 유틸리티로, 특정 검색조건에 해당하는 내용을 원하는 여러 Trace 파일에서 통합해서 한 파일로 만들어 준다. AP 서버를 두고 있는 분산 트랜잭션에서 여러 AP 서버의 Trace에 분산되어 있는 Trace 정보에서 원하는 검색조건인 클라이언트 식별자, 서비스명, 모듈명 또는 액션명 등으로 찾아볼 수 있다. Application End to End Tracing의 기능에 의해 여러 파일에 흩어져 있는 Trace 내용을 모으는데 유용하다.

• Automatic Optimizer Statistics Collection

각 오브젝트들에 대한 옵티마이저 통계정보를 자동으로 수집한다. 통계정보가 맞지 않거나 없는 경우에 자동으로 통계정보를 수집할 수 있다. DBA는 어떤 오브젝트에 대해서 통계정보를 실행해야 되는지, 어떤 통계정보가 맞지 않는지에 대해서 신경을 쓰지 않아도 되며, 직접 실행할 필요도 없다. 옵티마이저 통계정보는 GATHER_STATS_JOB로 자동으로 수집할 수 있으며, DB를 생성하거나 업그레이드할 때 자동으로 설정된다(디폴트). 이 기능은 Missing Statistics(통계정보가 없음) 또는 Stale Statistics(대량의 데이터 로딩 등에 의해 로우들이 10% 이상 변경된 경우)인 경우에 이들 오브젝트들을 관리하고 이들 오브젝트에 대해서 자동으로 수집한다. 수집 시기도 시스템 자원이 한가한 새벽시간에 운영된다.

지금까지 2회에 걸쳐 Oracle Optimizer의 원리 이해를 기반으로 SQL과 애플리케이션의 튜닝 방법에 대해서 설명하였다. 애플리케이션 튜닝 부분은 자세히 다루지는 않았지만, 이번 기회에 Oracle Optimizer의 원리를 이해하고, SQL 튜닝기법을 활용하여 시스템의 안정적인 개발 및 운영에 도움이 되었으면 하는 바람이다. ☺