오라클 성능
트러블슈팅의 기초

조동욱

㈜엑셈

ㄴ 서문 ㄱ

필자는 지난 1 여년간 오라클 성능 문제를 트러블슈팅하는데 필요한 기본 개념과 테크닉을 집대성하는데 많은 노력을 기울여왔습니다. 비록 최초의 원대한 계획에는 미치지 못하는 초라한 결과만을 남겼지만, 그 덕분에 이렇게 또 한 권의 책을 낼 수 있을 정도의 성과를 낼 수 있었습니다.

이 책은 제목 그대로 오라클 성능 문제를 트러블슈팅하기 위해 필요한 기본적인 개념과 테크닉을 소개하고 있습니다. 이전에 필자가 집필한 책들과 중복되는 내용들은 최대한 제거하고 가능한 새로운 내용들로 채우기 위해 노력했습니다. 이전 책들보다도 더 실용적으로 활용될 수 있도록 실제로 활용 가능한 소스 코드를 가능한 많이 실었습니다. 그 중에는 많은 분들이 생소해 할 다양한 프로그래밍 기법들이 포함되어 있습니다. 필자의 욕심으로 불필요한 같은 내용들이 포함된 것은 아닌가 하는 우려가 되는 것이 사실입니다. 모든 분들에게는 아니더라도 일부에게라도 현장에서 발생하는 문제를 해결하는데 활용될 수 있었으면 하는 바램입니다.

더불어 이 책이 기폭제가 되어 많은 실력있는 엔지니어들에 의해 보다 실용적이고 체계적인 트러블슈팅 기법들이 발표되고, 그런 기법을 구체화한 스크립트나 툴들이 제공되기를 기대해 봅니다. 그리고 그런 과정을 통해 대한민국 오라클 엔지니어들의 수준이 다시 한번 업그레이드되는 계기가 되었으면 하는 바램입니다.

마지막으로 이 책을 내는데 끝까지 후원을 해주신 (주)엑셈의 조종암 사장님께 감사의 말씀을 전합니다. 이런 강력한 후원 덕에 부족한 실력에도 불구하고 많은 책을 낼 수 있었습니다. 이런 노력들이 더욱더 결실을 맺고, 새로운 사람들과 새로운 성과들로 이어지기를 기원합니다.

조용한 마을 용상골에서

조 동욱

contents

contents

contents

contents

contents

1장 기본 개념 및 툴

1장

기본 개념 및 툴

오라클 성능 문제를 트러블슈팅하는 원리는 매우 간단합니다. 그리고 그 원리를 현실 세계의 문제에 적용하는 것도 어렵지 않습니다. 그럼에도 불구하고 여전히 오라클 성능 문제를 트러블슈팅하는 것이 어렵게 느껴지는 것은 두 가지 이유에서 기인한다고 봅니다.

- 가장 큰 원인은 성능 트러블슈팅의 기본 개념 및 필수적인 툴에 대한 사용법이 체계적으로 정립되어 있지 않다는 것입니다.
- 또 하나의 원인은 이런 기본기 위에 실제 현실 세계에서의 경험이 어느 정도 필요하다는 것입니다.

필자가 이 책을 기획한 이유가 위의 두 가지 문제를 해소하는데 최소한의 도움을 드리기 위해서입니다. 오라클 성능 문제를 트러블슈팅하기 위해 어떤 개념을 이해해야 하며, 그 개념을 활용하기 위해 어떤 툴들을 사용할 수 있어야 하며, 그리고 그것이 현실 세계의 성능 문제와 어떻게 연결되는가를 다양한 사례를 통해 소개하는 것이 목적입니다.

[기본 개념 및 툴]이 이 책의 제 1 장이 된 이유가 바로 이것입니다. 필자의 지식과 경험에도 뚜렷한 한계가 있기 때문에 문제를 해결할 수 있는 전지전능한 방법을 소개하기는 불가능합니다. 하지만, 성능 트러블슈팅에 필요한 최소한의 기본적인 지식은 공유할 수 있을 것으로 기대합니다.

본 장에서는 다음과 같은 내용을 통해 오라클 성능 트러블슈팅에 필요한 기본 개념과 툴을 소개합니다.

- 스냅샷과 프로파일링
- SQL*Plus 스크립팅

- 딕셔너리 뷰
- 진단 이벤트
- 덤프
- PL/SQL 패키지
- 자바 저장 프로시저
- 정규식
- oradebug

스냅샷과 프로파일링

오라클 성능 트러블슈팅의 기본 중의 기본, 핵심 중의 핵심은 무엇일까요? 필자가 생각하는 정답은 바로 **데이터입**니다. 오라클 성능 문제를 트러블슈팅하는 과정은 **필요한 데이터를 체계적으로 수집하고, 수집한 데이터에 기반해서 문제를 해석하고 해결책을 찾는 일련의 과정**입니다. 즉, 데이터가 없이는 성능 문제를 논할 수 없습니다.

필자가 많은 수의 성능 문제를 다루면서 현장의 엔지니어들이 가장 어려워한다고 느낀 것 중 하나가 바로 데이터를 수집하는 체계적인 방법입니다. 어떤 데이터를 어떻게 수집할 것인가? 이것이 시작점이면서 가장 어렵습니다.

필자는 오라클 성능 문제를 해석하는데 필요한 데이터를 **스냅샷 데이터**와 **프로파일링 데이터**의 두 가지로 분류합니다.

- **스냅샷 데이터 :** 특정 시점의 상태 데이터를 의미합니다. 가령 오늘 아침 10 시에 V$SESSTAT 뷰를 저장했다면, V$SESSTAT 뷰에 대한 스냅샷 데이터를 가지고 있다라고 말할 수 있습니다.

- **프로파일링 데이터 :** 특정 구간에서 수행된 작업에 대한 데이터를 의미합니다. 가령 오늘 아침 10 시부터 10 시 10 분까지 SQL*Trace 를 수행했다면, 10 분간의 프로파일링 데이터를 가지고 있다라고 말할 수 있습니다. 프로파일링 대신 **트레이스**라는 용어를 사용하기도 합니다.

스냅샷 데이터는 공간적인 개념입니다. 반면에 프로파일링 데이터는 시간적인 개념입니다. 이 두 가지 종류의 데이터를 합쳐야 완벽한 그림을 그릴 수 있습니다.

오라클이 제공하는 스냅샷 데이터를 정리해보면 다음과 같습니다.

- **대부분의 딕셔너리 뷰들 :** V$SESSTAT 뷰나 V$SESSION_WAIT 뷰와 같은 뷰들은 인스턴스의 현재 상태에 대한 정보를 제공합니다. 따라서 이들 뷰들은 스냅샷 데이터로 분류할 수 있습니다.

- **AWR 뷰들 :** AWR 이 제공하는 대부분의 뷰들은 특정 시점의 스냅샷 데이터들입니다. 오라클은 한시간에 한번씩 데이터베이스의 전반적인 상태에 대한 스냅샷을 생성합니다. AWR 스냅샷은 많은 수의 뷰들에 대한 스냅샷 데이터로 구성됩니다.

- **덤프 파일들 :** 오라클이 제공하는 덤프 파일들은 대부분 스냅샷 데이터입니다. 예를 들어 PGA 힙 덤프(PGA Heap Dump)는 PGA 의 현재 상태를 나타내는 스냅샷 데이터입니다.

오라클이 제공하는 프로파일링 데이터를 정리해보면 다음과 같습니다.

- **진단 이벤트들 :** SQL 문장의 수행 시간 정보를 기록하는 10046 진단 이벤트, 옵티마이저가 SQL 문장을 최적화하는 과정을 기록하는 10053 진단 이벤트 등은 모두 프로파일링 데이터를 생성합니다. 이들 데이터들은 프로세스가 하는 일을 시간 순으로 기록한 것들입니다.

- **매뉴얼 프로파일링 :** V$SESSION_WAIT 뷰 자체는 스냅샷 데이터에 해당합니다. 하지만 이 뷰를 반복적으로 액세스하면 마치 프로파일링을 수행한 것과 비슷한 효과를 얻을 수 있습니다.

스냅샷 데이터에서 의미있는 정보를 얻는 방법은 **차이**(Delta) 값을 구하는 것입니다. 특정 시점에서의 값 그 자체보다는 시점 A 와 시점 B 사이에 어떤 차이가 있는가가 핵심적인 데이터입니다. 가령 Logical Reads 가 얼마나 증가하는지, 리두 크기는 얼마나 증가하는지 등의 데이터를 얻는 것이 중요합니다. AWR 리포트를 사용해 보신 분이라면 그 의미를 직관적으로 파악하실 수 있을 것입니다. AWR 리포트는 두 개의 스냅샷 간의 차이(Delta) 값을 효과적으로 보여주는 것을 목적으로 합니다.

프로파일링 데이터에서 의미있는 정보를 얻는 방법은 **집계**(Summary)를 수행하는 것입니다. 프로파일링 데이터는 시간 순으로 모든 작업, 혹은 아주 많은 수의 작업을 기록합니다. 이 많은 데이터를 일일이 확인하는 것은 어렵습니다. 대신 효과적으로 집계를 함으로써 의미있는 정보를 얻게 됩니다. TKPROF 리포트를 사용해 보신 분이라면 그 의미를 직관적으로 이해하실 수 있을 것입니다. SQL 트레이스를 수행하면 시간순으로 모든

종류의 작업을 기록하지만, 이 정보를 직접 사용하기보다는 TKPROF 를 사용해서 집계 리포트를 만들어서 사용하게 됩니다.

위의 설명을 간략하게 요약하면 다음과 같습니다.

- 오라클 성능 트러블슈팅에 필요한 데이터는 **스냅샷** 데이터와 **프로파일링** 데이터로 나눌 수 있습니다.
- 스냅샷 데이터는 특정 시점의 상태에 대한 데이터를 의미하며, **차이**(Delta) 값을 통해서 의미있는 정보를 얻습니다.
- 프로파일링 데이터는 특정 시간 구간 내의 작업에 대한 데이터를 의미하며, **집계**(Summary) 값을 통해 의미있는 데이터를 얻습니다.

이제 구체적인 사례들을 통해서 스냅샷 데이터와 프로파일링 데이터를 얻는 방법을 소개하겠습니다.

스냅샷 데이터

간단한 형태의 스냅샷과 리포트

스냅샷 데이터와 리포트를 만드는 기본적인 방법은 다음과 같습니다.

- 시점 A 에서 스냅샷 A 를 만듭니다.
- 작업을 수행합니다.
- 시점 B 에서 스냅샷 B 를 만듭니다.
- 스냅샷 B 와 스냅샷 A 의 차이(스냅샷 B – 스냅샷 A)를 얻습니다.

간단한 예제를 통해 구체적인 방법을 알아보겠습니다.

우선 다음과 같이 V$SYSSTAT 뷰에 대한 스냅샷을 만듭니다. V$SYSSTAT 뷰의 내용을 테이블 SYSSTAT1 에 저장함으로써 스냅샷을 만들 수 있습니다.

```
SQL> create table sysstat1 as
```

```
  2  select * from v$sysstat;

Table created.
```

그리고 ALL_OBJECTS 뷰를 읽는 작업을 수행합니다.

```
SQL> select count(*) from all_objects;

  COUNT(*)
----------
     72795
```

다음으로 두번째 스냅샷을 테이블 SYSSTAT2 에 만듭니다.

```
SQL> create table sysstat2 as
  2  select * from v$sysstat;

Table created.
```

위에서 스냅샷 자체의 데이터는 큰 의미가 없으며, 두 스냅샷간의 **차이(Delta) 값이 중요**하다고 언급한 바 있습니다. 스냅샷간의 차이 값이야말로 두 개의 스냅샷에 해당하는 구간(시작 스냅샷 ~ 끝 스냅샷)에서 어떤 일이 발생했는지를 말해주기 때문입니다. 즉, 아래와 같이 두 개의 스냅샷, 테이블 SYSSTAT2 와 SYSSTAT1 간의 차이(Delta) 값을 구함으로써 어떤 변화가 있었는지 알 수 있습니다.

```
SQL> -- get delta
SQL> col name format a40
SQL> col delta format 999,999,999,999
SQL> select
  2     s1.name, (s2.value - s1.value) as delta
  3  from
  4     sysstat1 s1,
  5     sysstat2 s2
  6  where
  7     s1.statistic# = s2.statistic#
```

```
  8     and (s2.value - s1.value) > 0
  9  order by
 10     2 desc
 11  ;

NAME                                                  DELTA
----------------------------------------- ----------------
cell physical IO interconnect bytes             7,757,312
physical read total bytes                       7,618,560
physical read bytes                             7,585,792
session uga memory max                          2,131,824
session pga memory                              1,966,080
session pga memory max                          1,949,052
session uga memory                                553,868
physical write total bytes                        138,752
index scans kdiixs1                                95,559
buffer is pinned count                             84,072
table scan rows gotten                             81,313
physical write bytes                               57,344
... (중략) ...
switch current to new buffer                            1
redo synch writes                                       1

111 rows selected.
```

위의 스냅샷 리포트로부터 얻을 수 있는 결론 중 하나는 ALL_OBJECTS 뷰를 읽는 작업에 의해 Physical Reads 가 7M 바이트 이상 발생한다는 것입니다. 아주 간단한 방법으로 성능에 관한 중요한 사실들을 얻을 수 있습니다.

위의 예에서는 두 개의 스냅샷만을 다루고 있습니다. 즉, 두 개의 스냅샷간의 차이 값만을 리포팅합니다. 하지만 상황에 따라서는 **세 개 이상의 스냅샷간의 차이**를 비교해야 할 경우도 있습니다. 가령 1 분 간격으로 총 다섯개의 스냅샷을 생성했다면 차이 값은 총 4 개가 존재할 것입니다. 복잡한 성능 문제를 다룰 때는 이렇게 두 개 이상의 스냅샷을 필요로 하는 경우가 종종 있습니다. 다음 장들에서 몇 가지 예들을 보게 될 것입니다.

위의 예에서는 V$SYSSTAT 뷰에 대한 스냅샷만을 다루었지만, 성능 트러블슈팅을 위해서는 보다 많은 수의 뷰들에 대한 스냅샷 데이터가 필요합니다. V$SYSSTAT 뷰에 대한 스냅샷은 일량(Workload)에 대한 정보만을 제공합니다. 만일 대기 이벤트에 대한 정보

가 필요하다면 V$SYSTEM_EVENT 뷰를 스냅샷으로 사용할 수 있습니다. 래치 활동에 대한 정보가 필요하다면 V$LATCH 뷰를 스냅샷으로 저장하면 됩니다. 어떤 뷰라도 성능 문제에 대한 데이터만 제공한다면 스냅샷의 대상이 될 수 있습니다. 예를 들어 필자가 개인적으로 사용하는 라이브러리에서는 총 10 여개의 뷰에 대해 스냅샷을 생성합니다. AWR 스냅샷은 수십 개 이상의 뷰에 대한 스냅샷 데이터를 포함합니다.

AWR 스냅샷과 리포트

스냅샷 데이터에 기반한 리포트 중 가장 광범위한 데이터를 제공하는 것이 AWR 리포트입니다. AWR 리포트를 생성하는 방법 또한 앞서 설명한 것과 완벽하게 동일합니다.

- 시점 A 에서 AWR 스냅샷 A 를 생성합니다.
- 특정 작업을 수행합니다.
- 시점 B 에서 AWR 스냅샷 B 를 생성합니다.
- 스냅샷 B 와 스냅샷 A 의 차이(스냅샷 B – 스냅샷 A)를 얻습니다.

AWR 스냅샷은 1 시간마다 한번씩 자동으로 생성됩니다. 하지만 DBMS_WORKLOAD_REPOSITORY 패키지를 이용해서 수동으로 생성할 수도 있습니다. AWR 스냅샷을 생성하고, 스냅샷간의 차이 값을 리포팅하는 방법을 보겠습니다.

우선 다음과 같이 DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT 함수를 이용해 스냅샷 A 를 만듭니다.

```
SQL> -- 1st snapshot
SQL> col begin_snap new_value begin_snap;
SQL> col db_id new_value db_id;
SQL> col inst_num new_value inst_num;
SQL>
SQL> select dbid as db_id from v$database;

     DB_ID
----------
 864753741

SQL> select instance_number as inst_num from v$instance;
```

```
  INST_NUM
----------
         1

SQL>
SQL> select dbms_workload_repository.create_snapshot as begin_snap from dual;

BEGIN_SNAP
----------
      2874
```

그리고 ALL_OBJECTS 뷰를 읽는 작업을 수행합니다.

```
SQL> select count(*) from all_objects;

  COUNT(*)
----------
     72796
```

스냅샷 B 를 만듭니다.

```
SQL> -- 2nd snapshot
SQL> col end_snap new_value end_snap;
SQL> select dbms_workload_repository.create_snapshot as end_snap from dual;

  END_SNAP
----------
      2875
```

그리고 DBMS_WORKLOAD_REPOSITORY.AWR_REPORT_TEXT[_HTML] 함수를 이용해서 스냅샷 B 와 스냅샷 A 간의 차이를 리포팅합니다.

```
SQL> --- AWR report
SQL> select * from table(
  2     dbms_workload_repository.awr_report_text(
```

```
  3          &db_id,
  4          &inst_num,
  5          &begin_snap,
  6          &end_snap)
  7    );

... (중략) ...
Top 5 Timed Foreground Events
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                           Avg
                                                          wait   % DB
Event                                 Waits     Time(s)   (ms)   time Wait Class
------------------------------ ------------ ----------- ------ ------ ----------
DB CPU                                              5           75.3
db file sequential read                173           1      8   18.1 User I/O
asynch descriptor resize            26,068           1      0    7.7 Other
control file sequential read           105           0      2    2.4 System I/O
direct path read                         5           0      9     .7 User I/O
... (중략) ...
```

너무나 복잡하고 방대해 보이는 AWR 데이터도 **스냅샷이라는 관점**에서 보면 매우 보편적이고 평범한 데이터에 불과하다는 것을 알 수 있습니다. 다만, 어떤 뷰들을 스냅샷의 대상으로 할 것인가, 그리고 스냅샷간의 차이 값을 어떻게 보여줄 것인가에 따라 매우 다양한 방식으로 사용할 수 있다는 차이가 있을 뿐입니다.

덤프와 스냅샷

오라클 성능 트러블슈팅을 위해 필요한 스냅샷 데이터의 99%는 오라클이 제공하는 동적 성능 뷰(Dynamic Performance View)에서 얻을 수 있습니다. 위에서 예제로 설명한 데이터들도 모두 동적 성능 뷰에 기반하고 있습니다. 하지만 나머지 1% 정도의 문제에 대해서는 동적 성능 뷰만으로 부족한 경우가 있습니다.

가장 대표적인 경우가 PGA 의 크기가 계속 증가하면서 세션의 성능이 저하되는 현상입니다. 이 경우 V$SESSTAT 뷰와 같은 동적 성능 뷰를 통해서 얻을 수 있는 것은 PGA 의 크기가 매 스냅샷마다 얼마나 증가하느냐입니다. 가령 1 분 간격으로 5 개의 스냅샷을 생성했는데, 각 스냅샷마다 PGA 의 크기가 100M 바이트씩 증가(즉 각 스냅샷간의 차이

값이 100M 바이트)한다는 사실을 알 수 있습니다. 하지만 왜 크기가 증가하는지의 이유는 알 수 없습니다. 크기가 커졌다는 사실만 알 수 있고, 왜 크기가 증가했는지를 알 수 없다면 트러블슈팅의 궁극적인 목적인 해결책을 만드는 것이 불가능합니다.

이런 경우에 사용할 수 있는 것이 PGA 힙 덤프입니다. 가령 아래와 같은 명령으로 PGA 힙 덤프를 수행합니다.

```
SQL> -- every dump is snapshot data!!!
SQL> alter session set events 'immediate trace name heapdump level 1';

Session altered.
```

위의 명령을 수행하면 현재 프로세스의 PGA 힙 정보가 트레이스 파일에 기록됩니다.

```
HEAP DUMP heap name="pga heap"  desc=0FB8A628
 extent sz=0x206c alt=108 het=32767 rec=0 flg=2 opc=2
 parent=00000000 owner=00000000 nex=00000000 xsz=0xfff8 heap=00000000
 fl2=0x60, nex=00000000
EXTENT 0 addr=0E950008
  Chunk  e950010 sz=    12712    perm      "perm           "  alo=12712
  Chunk  e9531b8 sz=     3300    freeable  "diag pga       "  ds=0D3B04C0
  Chunk  e953e9c sz=     8460    perm      "perm           "  alo=4224
  Chunk  e955fa8 sz=    32796    freeable  "kgh stack      "
  Chunk  e95dfc4 sz=     8252    freeable  "Alloc environm "  ds=0DD6C880
EXTENT 1 addr=0FFC0008
  Chunk  ffc0010 sz=     2548    free      "               "
  Chunk  ffc0a04 sz=     8588    freeable  "Alloc environm "  ds=0DD6C880
  Chunk  ffc2b90 sz=     1292    freeable  "koh-kghu call h"
... (중략) ...
```

PGA 힙 덤프와 같은 덤프 데이터는 가장 자세한 형태의 스냅샷 데이터라고 볼 수 있습니다. 이런 종류의 스냅샷 데이터는 지나치게 정밀하기 때문에 다른 종류의 스냅샷 데이터처럼 차이(Delta) 값을 구하는 것이 어렵기도 할뿐더러 큰 의미가 없습니다. 오히려 **적절한 형태로 집계**할 필요가 있습니다. 제 4 장 [힙 메모리 분석]에서 상세한 방법을 소개할 것입니다.

앞서 언급한 것처럼 오라클 성능 트러블슈팅을 위해 덤프를 수행하고 그 결과를 분석하는 것은 1%의 특별한 경우에만 사용되는 방법입니다. 하지만 이 1%를 해결할 수 있느냐 없느냐가 무엇보다 중요한 경우가 있습니다. 빈도로는 1%이지만 중요도로는 10% 이상이라고 생각하시면 좋겠습니다.

프로파일링 데이터

간단한 형태의 매뉴얼 프로파일링

프로파일링 데이터를 얻고 리포팅하는 기본적인 방법은 다음과 같습니다.

- 프로파일링 대상이 되는 데이터(주로 딕셔너리 뷰)를 필요한 만큼 반복적으로 액세스해서 읽습니다.
- 위와 같이 읽은 데이터를 원본 데이터로 이용할 수 있습니다.
- 원본 데이터를 적절히 집계해서 리포팅함으로써 보다 의미있는 데이터를 얻을 수 있습니다.

위의 설명만으로는 의미 전달이 쉽지 않을 것입니다. 간단한 예를 통해 보다 구체적으로 설명하겠습니다.

세션 #1 에서 ALL_OBJECTS 뷰를 읽는 작업을 수행합니다.

```
SQL> -- session #1
SQL> exec dbms_application_info.set_client_info('session1');

PL/SQL procedure successfully completed.

SQL> select count(*) from all_objects;
... (수행 중) ...
```

세션 #1 이 작업을 수행하는 동안 세션 #2 로부터 세션 #1 이 어떤 대기 이벤트(Wait Event)를 대기하는지 프로파일링하는 것이 이번 예제의 핵심입니다. 우선 세션 #2 에서 세션 #1 의 세션 아이디(SID) 값을 얻습니다.

```
SQL> -- session #2
SQL> col sid new_value sid
SQL> select sid from v$session where client_info = 'session1';

       SID
----------
       141
```

그리고 총 1,000 회에 걸쳐 **V$SESSION_WAIT** 뷰를 반복적으로 읽으면서 대기 이벤트 정보를 얻습니다. 즉 **대기 이벤트 정보를 프로파일링**합니다.

```
SQL> set serveroutput on
SQL> declare
  2     v_sw      v$session_wait%rowtype;
  3  begin
  4     for idx in 1 .. 1000 loop
  5             begin
  6                       select * into v_sw
  7                       from v$session_wait
  8                       where sid = &sid
  9                                 and state = 'WAITING';
 10
 11                       dbms_output.put_line(
 12                                 'event = ' || v_sw.event ||
 13                                 ', seq# = ' || v_sw.seq# ||
 14                                 ', p1 = ' || v_sw.p1 ||
 15                                 ', elapsed = ' ||
trunc(v_sw.wait_time_micro/1000,2) || '(ms)');
 16             exception when others then
 17                       null;
 18             end;
 19     end loop;
 20  end;
 21  /
```

그 결과로 아래와 같이 대기 이벤트 목록을 얻을 수 있습니다.

```
... (중략) ...
event = direct path read, seq# = 63693, p1 = 1, elapsed = 11.55(ms)
event = direct path read, seq# = 63693, p1 = 1, elapsed = 11.64(ms)
event = direct path read, seq# = 63693, p1 = 1, elapsed = 11.71(ms)
event = asynch descriptor resize, seq# = 63694, p1 = 1, elapsed = .07(ms)
event = asynch descriptor resize, seq# = 63694, p1 = 1, elapsed = .15(ms)
event = asynch descriptor resize, seq# = 63694, p1 = 1, elapsed = .21(ms)
... (중략) ...
```

V$SESSION_WAIT 뷰 자체는 스냅샷 데이터에 해당하지만, 그 뷰를 시간의 흐름에 따라 반복적으로 액세스하면 프로파일링 데이터가 됩니다. 즉, **프로파일링 데이터는 스냅샷 데이터의 시간 확장판**이라고도 할 수 있습니다.

프로파일링 데이터는 **집계**(Summary)를 수행함으로써 보다 의미있는 데이터를 얻을 수 있습니다. 가령 위와 같은 방법으로 얻은 프로파일링 데이터에 대해 대기이벤트를 기준으로 히트(Hit) 회수를 집계한다면 어떤 대기 이벤트가 주로 발생했는지를 알 수 있을 것입니다. 이를 위해서 아래와 같은 방법을 사용할 수 있습니다.

우선, 프로파일링 데이터를 저장할 테이블 SESSION_WAIT 를 만듭니다.

```
SQL> create table session_wait
  2  as
  3  select *
  4  from v$session_wait
  5  where 1 = 0
  6  ;

Table created.
```

세션 #1 에 대해 대기 이벤트를 프로파일링하면서 그 결과를 테이블 SESSION_WAIT 에 저장합니다.

```
SQL> begin
```

```
  2     for idx in 1 .. 1000 loop
  3
  4               insert into session_wait
  5               select *
  6               from v$session_wait
  7               where sid = &sid
  8                         and state = 'WAITING';
  9
 10               dbms_lock.sleep(0.001);
 11
 12     end loop;
 13  end;
 14  /

PL/SQL procedure successfully completed.
```

프로파일링 결과가 테이블에 저장되어 있기 때문에 다음과 같이 손쉽게 집계 리포트를 만들 수 있습니다.

```
SQL> select
  2     event,
  3     count(*) as hits
  4  from
  5     session_wait
  6  group by
  7     event
  8  order by
  9     2 desc
 10  ;

EVENT                                                                HITS
---------------------------------------------------------------- ----------
asynch descriptor resize                                             11
db file sequential read                                               1
```

위의 방법을 응용하면 테이블을 만들고 테이블에 INSERT 하는 등의 번거로운 작업 없이 하나의 쿼리로 동일한 결과의 리포트를 만들 수 있습니다. 가령 아래와 같은 형태의 쿼리를 사용할 수 있습니다.

```
SQL> select /*+ no_query_transformation
  2              ordered use_nl(w) */
  3     w.event,
  4     count(*) as hits
  5  from
  6     (select level from dual connect by level <= 50000) x,
  7     (select
  8              decode(state, 'WAITING', event, 'ON CPU') as event
  9     from v$session_wait
 10     where sid = &sid) w
 11  group by
 12     w.event
 13  order by
 14     2 desc
 15  ;
```

위의 쿼리를 해석하는 방법은 다음과 같습니다.

- DUAL 테이블에서 50,000 건을 읽는 뷰 X 가 선행 테이블이 됩니다.
- 뷰 X 를 선행 테이블로 해서 V$SESSION_WAIT 뷰를 Nested Loops Join 으로 읽습니다.
- 위와 같이 읽은 결과를 대기 이벤트(w.event)별로 집계합니다.

위와 같은 방식으로 쿼리가 수행되면 **V$SESSION_WAIT** 뷰를 50,000 번 반복적으로 읽으면서 프로파일링하고(Nested Loops Join 에 의해), 그 결과를 집계(Group By 에 의해)하는 효과가 있습니다. 단, 위와 같은 효과를 얻기 위해서 다음과 같은 힌트를 사용하고 있습니다.

- NO_QUERY_TRANSFORMATION 힌트를 사용해서 뷰 머지(View Merge)가 발생하는 것을 막습니다. 뷰 머지가 발생하면 테이블의 조인 순서가 예상치 못하게 바뀔 수 있기 때문입니다.
- ORDERED 힌트를 이용해서 뷰 X 가 선행 테이블(아우터 테이블)이 되게 합니다.
- USE_NL 힌트를 이용해서 루프를 돌면서 반복적으로 후행 테이블(이너 테이블) V$SESSION_WAIT 뷰를 읽도록 합니다.

위와 같은 방식으로 쿼리를 수행함으로써 아래와 같이 **프로파일링과 집계가 동시에 수행**된 결과를 얻을 수 있습니다.

```
EVENT                          HITS
------------------------------ ----------
ON CPU                          30529
asynch descriptor resize        15617
direct path read                 3854
```

위의 쿼리를 조금 더 세련되게 다듬으면 대기 이벤트별 대기 회수와 대기 시간도 어느 정도 추측할 수 있습니다. 아래에 예제가 있습니다.

```
SQL> -- session #2
SQL> col event format a30
SQL> with w as (
  2     select /*+ no_query_transformation
  3                         ordered use_nl(w) */
  4              w.event,
  5              count(*) as hits,
  6              count(distinct seq#) as dist_waits,
  7              50000 as total_hits
  8     from
  9              (select level from dual connect by level <= 50000) x,
 10              (select
 11                        decode(state, 'WAITING', event, 'ON CPU') as event,
 12                        decode(state, 'WATIING', seq#, 0) as seq#
 13              from v$session_wait
 14              where sid = &sid) w
 15     group by
 16              w.event
 17     order by
 18              2 desc
 19  )
 20  select /*+ ordered */
 21     w.event,
 22     trunc(100*w.hits/w.total_hits,1) as wait_pct,
 23     trunc((t2.gt - t1.gt)*w.hits/w.total_hits/100,2) as wait_time,
 24     trunc((t2.gt - t1.gt)*w.hits/w.total_hits/w.dist_waits/100,3) as avg_wait_time
 25  from
 26     (select dbms_utility.get_time as gt from dual) t1,
 27     w,
 28     (select dbms_utility.get_time as gt from dual) t2
 29  order by
 30     2 desc
```

```
31  ;
```

위의 쿼리를 해석하는 방법은 다음과 같습니다.

- V$SESSION_WAIT 뷰를 읽는 방식은 동일합니다. V$SESSION_WAIT 뷰를 후행 테이블로 해서 Nested Loops Join 을 수행함으로써 반복적으로 프로파일링을 수행하게 합니다. 그리고 그 결과를 대기 이벤트에 대해 집계함으로써 대기 이벤트별 히트 회수, 대기 회수 등의 정보를 얻습니다.
- 위의 작업을 수행하기 전 후에 각각 현재 시간을 얻음으로써 시작 시간과 끝 시간을 얻습니다.
- 이렇게 얻은 정보를 잘 조합하면 대기 이벤트별 대기 시간을 추측할 수 있습니다.

아래에 그 결과가 있습니다.

```
EVENT                          WAIT_PCT  WAIT_TIME AVG_WAIT_TIME
------------------------------ ---------- ---------- -------------
ON CPU                             91.8        .53          .532
asynch descriptor resize            8.1        .04          .047
```

제 3 장 [대기 이벤트 분석]에서 보다 세련된 방법으로 프로파일링을 수행하고 집계하는 방법을 소개할 것입니다.

위에서 소개한 매뉴얼 프로파일링 기법은 V$SESSION_WAIT 뷰나 V$SESSION 뷰와 같은 매 순간마다 정보가 바뀌는 뷰들(즉, 스냅샷 데이터)에 대해서 최소한의 오버헤드로 가능한 많은 회수의 프로파일링 데이터를 얻고 이를 집계할 수 있다는 점에서 매우 유용하다고 하겠습니다. 하지만, 아무리 자주 액세스한다고 하더라도 **샘플링(Sampling)의 한계**를 벗어나기는 어렵다는 점도 명확하게 인식해야 합니다.

진단 이벤트와 프로파일링

위에서 소개한 매뉴얼 프로파일링 기법의 한계는 100% 완벽한 데이터가 아니라는 점입니다. V$SESSION_WAIT 뷰를 아무리 자주 액세스한다고 하더라도 수집하지 못하는 시점의 데이터가 있기 마련입니다. 반면에 오라클이 자체적으로 제공하는 프로파일링 데이터는 이런 제약이 없습니다.

가장 대표적인 예가 **10046 진단 이벤트**입니다. 다른 용어로는 **Extended SQL*Trace** 라고도 부릅니다. 10046 진단 이벤트의 정확한 목적은 SQL 수행의 각 단계별로 시간 정보를 얻는 것입니다.

```
prompt> oerr 10046
 "enable SQL statement timing"
// *Cause:
// *Action:
```

프로파일링 데이터는 시계열 데이터라는 것을 앞서 언급한 바가 있죠? 이러한 요구 사항을 가장 완벽하게 구현한 것이 **10046 진단 이벤트**입니다. 즉, 10046 진단 이벤트는 프로파일링을 수행하는 가장 완벽한 방법입니다.

가령 아래와 같은 방법으로 대기 이벤트의 대기 시간까지 포함된 프로파일링 데이터를 얻을 수 있습니다.

```
SQL> alter session set events '10046 trace name context forever, level 8';

Session altered.

SQL> select count(*) from all_objects;

  COUNT(*)
----------
     72796

SQL> alter session set events '10046 trace name context off';

Session altered.
```

10046 진단 이벤트에 의해 생성된 프로파일링 데이터는 서버 프로세스의 트레이스 파일에 기록됩니다. 아래 데이터를 보면 앞서 매뉴얼 프로파일링을 통해서 얻은 데이터와 거의 유사한 포맷으로 기록되는 것을 알 수 있습니다.

```
PARSING IN CURSOR #12 len=32 dep=0 uid=97 oct=3 lid=97 tim=271686695607 hv=789896629
ad='16c89bb8' sqlid='9tz4qu4rj9rdp'
select count(*) from all_objects
END OF STMT
PARSE
#12:c=156250,e=188203,p=1,cr=552,cu=0,mis=1,r=0,dep=0,og=1,plh=1001908815,tim=2716866
95604
EXEC #12:c=0,e=155,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1001908815,tim=271686695918
WAIT #12: nam='SQL*Net message to client' ela= 6 driver id=1413697536 #bytes=1 p3=0
obj#=85351 tim=271686696060
WAIT #12: nam='direct path read' ela= 36545 file number=1 first dba=241 block cnt=7
obj#=18 tim=271686735654
...
```

프로파일링 데이터는 **집계**(Summary)를 통해서 의미있는 정보를 얻을 수 있습니다. 10046 진단 이벤트에 의해 생성된 프로파일링 데이터는 **TKPROF** 툴을 이용해 집계 리포트를 얻을 수 있습니다.

```
prompt> tkprof xxx.trc xxx.out

SQL ID: 9tz4qu4rj9rdp
Plan Hash: 1001908815
select count(*)
from
 all_objects


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      6.40       6.92        925      44249          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      6.40       6.92        925      44249          0           1
```

```
... (중략) ...
```

TKPROF 는 Transient Kernel Profiler 의 약자입니다. Profiler 라는 용어에 주목하시기 바랍니다. TKPROF 가 다루는 데이터가 프로파일링 데이터라는 것을 스스로 말하고 있습니다.

10053 진단 이벤트는 옵타마이저가 최적화(Optimization)를 수행하는 과정을 시간 순으로 기록합니다. 즉, 쿼리 최적화라는 작업을 프로파일링합니다. 10053 진단 이벤트의 정의는 다음과 같습니다.

```
prompt> oerr 10053
"CBO Enable optimizer trace"
// *Cause:
// *Action:
```

진단 이벤트의 정의로부터 프로파일링을 수행한다는 것을 명확하게 알 수 있습니다.

10053 진단 이벤트에 의해 생성되는 프로파일링 데이터를 직접 확인해 보겠습니다. 우선 아래와 같이 10053 진단 이벤트를 활성화한 상태에서 쿼리를 수행합니다.

```
SQL> alter session set events '10053 trace name context forever, level 1';

Session altered.

SQL> select count(*)
  2  from all_objects
  3  where rownum = 1
  4  ;

  COUNT(*)
----------
         1

SQL> alter session set events '10053 trace name context off';

Session altered.
```

그리고 트레이스 파일의 내용을 확인해보면 옵티마이저가 수행하는 일련의 작업이 시간 순으로 완벽하게 기록된 것을 알 수 있습니다. 이렇게 기록된 데이터는 "왜 이런 실행 계획이 만들어지는가?" 라는 의문을 해결할 수 있는 가장 완벽한 정보를 제공합니다.

```
SQL> @trace_file
------------------------------------------------------------------------
c:\oracle\diag\rdbms\ukja1120\ukja1120\trace\ukja1120_ora_1796.trc

... (중략) ...
***************************************
GENERAL PLANS
***************************************
Considering cardinality-based initial join order.
Permutations for Starting Table :0
Join order[1]:  OBJAUTH$[OBJAUTH$]#0  X$KZSRO[X$KZSRO]#1

***************
Now joining: X$KZSRO[X$KZSRO]#1
... (중략) ...
```

10053 진단 이벤트에 의해 생성된 프로파일링 데이터(트레이스 파일)는 원본 데이터를 직접 이용하는 것이 가장 보편적인 사용 방법입니다. 하지만 필요하다면 집계를 통해서 원하는 데이터를 얻을 수 있습니다. 예를 들어 아래와 같은 방법으로 트레이스 파일로부터 조인 회수를 구할 수 있습니다.

```
SQL> select count(*)
  2  from table(tpack.get_tracefile_contents(tpack.get_tracefile_name))
  3  where column_value like 'Now joining%'
  4  ;

  COUNT(*)
----------
       301
```

트레이스 파일을 PL/SQL 을 이용해서 읽고 해석하는 방법에 대해서는 이후에 상세하게 다루게 될 것입니다.