Books
정오표

[1권 정오표 (1쇄)]

페이지	원문	수정문
018 (그림 1-4) 239 (그림 3-18) 243 (그림 4-1) 345 (그림 5-1) 431 (그림 6-1)	Darabase Writer(DBW0)	Database Writer(DBWn)
	Archiver(ARC0)	Archiver(ARCn)
1장. 오라클 아키텍처		
039 (맨 상단)	① 사용자가 커밋을 날리면 ② LGWR는 커밋 레코드를 Redo 로그 버퍼에 기록하고③ 이것을 즉시 트랜잭션 로그 엔트리와 함께 redo 로그파일에 저장하고 나서 ④ 커밋을 수행한 트랜잭션에 "success code"를 리턴한다.	① 사용자가 커밋을 날리면 ② 서버 프로세스가커밋 레코드를 Redo 로그 버퍼에 기록하며, ③LGWR는 이것을 즉시 트랜잭션 로그 엔트리와 함께 redo 로그파일에 저장한 후 ④ 커밋을 수행한 서버 프로세스에 "success code"를 리턴한다.
062	읽는 중에 CR copy를 생성할 필요가 없어 Current 블록을 읽더라도 Consistent 모드에서 읽었다면 'guery' 항목에 집계된다.	읽는 중에 CR copy를 생성할 필요가 없어 Current 블록을 읽더라도 Consistent 모드에서 읽었다면 'query ' 항목에 집계된다.
081	이 프로그램이 돌기 시작한지 얼마 지나지 않아 다른 트랜잭션에 의해 홍길동 고객의수납액이 10,000원에서 20,000원으로 변경되고 나서 커밋되었다.	이 프로그램이 돌기 시작한지 얼마 지나지 않아 홍길동 고객의 수납액을 10,000원에서 20,000원으로 변경하고 나서 커밋하였다.
083 (그림 1-17)	쿼리 SCN 123	쿼리 SCN 90
2장. 트랜잭션과 Lock		
110	UPDATE 계좌 SET 잔고 = 잔고 - 50000 , 등급 = 'A' WHERE 계좌번호 = 123;	UPDATE 계좌 SET 잔고 = 잔고 - 50000 WHERE 계좌번호 = 123;
112 (맨 하단)	트랜잰션	트랜잭션
119 (첫 문장)	Lock에 때문에	Lock 때문에
121 (하단에 추가)	**중대한 버그** 본서가 출간되고 2쇄를 시작하기 직전, 방금 설명한 기능(ora_rowscn을 이용한 동시성 제어)에 중대한 버그(Bug 5270479)가 있음을 발견하였다. 트랜잭션이 아래 패턴 1과 같을	

	때는 정상적인 결과에 이르지만, 패턴 2(--> 두 트랜잭션이 update 를 동시에 진행)와 같을 때는 TX1 이 TX2 의 갱신을 덮어써 Lost Update 가 발생하는 버그다. < 패턴 1 > < 패턴 2 > --------------+----+-------------- ---------------+----+-------------- TX1 | | TX2 TX1 | | TX2 --------------+----+-------------- ---------------+----+-------------- select ... ; | t1 | select ... ; | t1 | | t2 | update ... ; | t2 | update ... ; | t3 | commit; update ... ; | t3 | update ... ; | t4 | | t4 | commit; --------------+----+-------------- ---------------+----+-------------- 안타까운 것은, 11gR2(11.2.0.1)에서도 Bug Fix 가 되지 않았다는 사실(Bug 7338384)이다. 결론적으로, (언제일지 모르지만) 버그가 고쳐질 때까지 이 기능을 이용한 동시성 제어를 할 수 없게 되었다. 버그 리포트(Bug 5270479)에서 오라클이 제시하는 대안은 다음과 같다. 즉, 기존 방식대로 동시성 제어를 하라는 뜻이다. "Include a sequence column in the table and increment that on every UPDATE then use that in predicates to ensure that the row has not been modified."

131	이제 TX Lock 을 획득했으므로 **트랜잭**을 위한 일련의 작업들을 수행할 수 있다.	이제 TX Lock 을 획득했으므로 **트랜잭션**을 위한 일련의 작업들을 수행할 수 있다.
3 장. 오라클 성능 관리		
168	각 단계에서 읽거나 **갱시한** 처리 건수	각 단계에서 읽거나 **갱신한** 처리 건수
173	이를 통해 CPU time 과 Elapsed time 간에 갭(GAP)이 발생했던 이유를 설명할 수 있다.	이를 통해 CPU time 과 Elapsed time 간**의** 갭(GAP)**, 그리고 사용자가 느낀 총 소요시간과의 갭**이 발생했던 이유를 설명할 수 있다.
187	select * from table(dbms_xplan.display_cursor('[sql_id]',[child_no],'[format)'));	select * from table(dbms_xplan.display_cursor('[sql_id]',[child_no],'[format]'));
192	작업을 수행할 **세션과는 별도의** 세션을 하나 더 열어 아래 create 문을 수행한다.	작업을 수행할 **세션과 별도로** 세션을 하나 더 열어 아래 create 문을 수행한다.
206	and t2.col between :range1 and **range2** ;	and t2.col between :range1 and **:range2** ;
210 (하단 주석)	오라클 8 이전에도 **ultbstat/utleatat** 스크립트를 이용해 비슷한 리포트를 뽑아볼 수 있었다.	오라클 8 이전에도 **utlbstat/utlestat** 스크립트를 이용해 비슷한 리포트를 뽑아볼 수 있었다.
224 (그림 위 문장)	이미 AWR 에 쓰여진 **된** 것이므로...	이미 AWR 에 쓰여진 것이므로...
226	본 **장에서** 설명하려는 ~~	본 **절에서** 설명하려는 ~~
238 (맨 하단)	이를 위해서는 다음 장부터 설명하는 원리들을 십분 **이해해고** 활용할 수 있어야만 한다.	이를 위해서는 다음 장부터 설명하는 원리들을 십분 **이해하고** 활용할 수 있어야만 한다.
325 (중하단)	인덱스 사용이 가능하지만 사용자가 :isu_cd 값을 입력하지 않았을 때 Table Full Scan 이 유리한데도 인덱스를 사용하게 되므로 성능이 나빠질 수 있다.	인덱스 사용이 가능하지만 사용자가 :isu_cd 값을 입력하지 않았을 때 Table Full Scan 이 유리한데도 인덱스를 사용하게 되므로 성능이 나빠질 수 있다. **또 주의할 점은, null 허용 컬럼일 때 결과집합이 달라질 수 있으므로 반드시 not null 컬럼일 때만 사용해야 한다는 것이다.**
326 (중간)	C 와 D 방식은, 사용자의 :isu_cd 입력 여부에 따라 Full Table Scan 과 Index	C 와 D 방식은, 사용자의 :isu_cd 입력 여부에 따라 Full Table Scan 과 Index Scan 으로

	Scan 으로 실행계획이 자동 분기된다. **단, nvl 또는 decode 함수를 사용할 때는** 해당 컬럼이 not null 컬럼이어야 하며, null 허용 컬럼일 때는 결과집합이 달라지므로 주의해야 한다.	실행계획이 자동 분기된다. **단, 앞서 설명한 like 연산자처럼 nvl 또는 decode 함수를 사용할 때도** 해당 컬럼이 not null 컬럼이어야 하며, null 허용 컬럼일 때는 결과집합이 달라지므로 주의해야 한다.
5 장. 데이터베이스 Call 최소화 원리		
354 (맨 하단)	Execute Call 이 최대 100 만 번, 따라서 최대 200 만 번의 데이터베이스 Call 이 발생하게 된다.	Execute Call 이 최대 500 만 번, 따라서 최대 600 만 번의 데이터베이스 Call 이 발생하게 된다.
375 (중하단)	TDU 는 1,048 로 설정했다고 가정하자. 그러면 총 17 개 패킷으로 단편화되어 클라이언트에게 전송이 이루어진다. fetch1 : (1,460 + 588) + (1,460 + 588) + 404 = 4,500 (-> 5 개 패킷) fetch2 : (1,460 + 588) + (1,460 + 588) + 404 = 4,500 (-> 5 개 패킷) fetch3 : (1,460 + 588) + (1,460 + 588) + 404 = 4,500 (-> 5 개 패킷) fetch4 : (1,460 + 1240) = 2,700 (-> 2 개 패킷)	TDU 는 1,024 로 설정했다고 가정하자. 그러면 총 18 개 패킷으로 단편화되어 클라이언트에게 전송이 이루어진다. fetch1 : (1,024 + 1,024) + (1,024 + 1,024) + 404 = 4,500 (-> 5 개 패킷) fetch2 : (1,024 + 1,024) + (1,024 + 1,024) + 404 = 4,500 (-> 5 개 패킷) fetch3 : (1,024 + 1,024) + (1,024 + 1,024) + 404 = 4,500 (-> 5 개 패킷) fetch4 : (1,024 + 1,024) + 652 = 2,700 (-> 3 개 패킷)
402~403	(403 페이지 첫 줄에 문자가 겹쳤음. 아래와 같이 수정함) 아래처럼 일반 조인문 또는 스칼라 서브쿼리를 사용할 때만 완벽한 문장수준 읽기 일**관성이 보장된다.** **# 일반 조인문** select a.지수업종코드 ……	
6 장. I/O 효율화 원리		
449 (맨 하단)	*db_block_size* 는 16 이고, Multiblock I/O 단위는 16 이다.	*db_block_size* 는 8,192 이고, Multiblock I/O 단위는 16 이다.
453 (맨 하단)	개별 쿼리의 수행 속도를 향상시키는 데 주로 도움을 준다.	**I/O 를 위한 시스템 Call 을 줄이고** 개별 쿼리의 수행 속도를 향상시키는 데 주로 도움을 준다.
460 (하단)	~~ 데이터파일에 직접 insert 하므로 일반적인 insert 와는 비교할 수 없을 정도로 빠르다. 게다가 **Direct Path Insert 에서는 Redo 와 Undo 엔트리를 로깅하지 않도록 옵션을 줄 수도 있어** 더 빠른 insert 가 가능하다.	~~ 데이터파일에 직접 insert 하므로 일반적인 insert 와는 비교할 수 없을 정도로 빠르다. HWM 바깥 영역에 데이터를 입력하므로 Undo 발생량도 최소화된다(HWM 뒤쪽에 입력한 데이터는 커밋하기 전까지 다른 세션에 읽히지 않으므로 Undo 데이터를 제공하지 않아도 되고, 롤백할 때는 할당된 익스텐트에 대한 딕셔너리 정보만 롤백하면 되기 때문). 게다가 Direct Path Insert 에서는 Redo 로그까지 최소화(데이터 딕셔너리 변경사항만 로깅)하도록 옵션을 줄 수도 있어 더 빠른 insert 가 가능하다.
부록		
495 (그림 7-1)	중간에 EMP 테이블(TYPE:TABLE)이 2 개	오른쪽은 EMP 가 아니라 DEPT 임

[1권 정오표 (3쇄)]

p.172-177 다음 내용 추가

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	11	0.03	0.05	18	39	0	1000
total	13	0.03	0.07	18	39	0	1000

1,000건을 읽는데 ArraySize를 100으로 설정했으므로 one-row fetch[26]까지 합쳐 11번의 Fetch Call이 발생했고, Parse Call과 Execute Call까지 합쳐 총 13번의 Call이 발생했다. 따라서 13번 발생한 Call 각각에 대한 소요시간을 더한 총 소요시간이 0.07초다. 데이터베이스 내부적으로는 0.07초 동안만 일을 했고, 나머지 시간은 애플리케이션으로부터 추가 Call을 기다리면서 Idle 상태로 대기한 시간이다.

또한, CPU time과 Elapsed time 간 시간 차는, 0.07초간 일하는 동안에도 실제로 프로세스가 CPU를 점유하고 원활하게 일을 진행한 시간은 0.03초에 불과하다는 사실을 말해 준다. 나머지 0.04초는 대기(Wait) 상태에 빠졌던 것으로 이해하면 된다. 지금까지 설명한 내용은 그림 3-1을 통해 쉽게 이해할 수 있다.

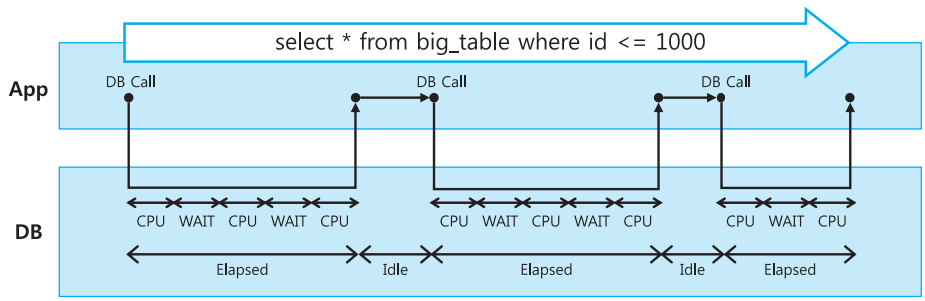

그림 3-1

26) SQL*Plus 등에서 첫 번째 Fetch는 ArraySize와 무관하게 한 건만 가져오는 것을 말하며, 5장 데이터베이스 Call 최소화 원리에서 설명한다.

SQL 수행 전 이벤트 트레이스 레벨을 8로 설정했으므로 다음과 같이 이벤트 발생 현황까지 확인할 수 있다. 이를 통해 CPU time과 Elapsed time 간의 갭(GAP), 그리고 사용자가 느낀 총 소요시간과의 갭이 발생했던 이유를 설명할 수 있다.

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       11        0.00          0.00
  db file sequential read                         18        0.02          0.02
  SQL*Net message from client                     11        1.43          8.43
  SQL*Net more data to client                     20        0.00          0.00
```

가장 눈에 띄는 것은 *SQL*Net message from client* 이벤트다. 이 대기 이벤트는 Idle 이벤트로서 오라클 서버 프로세스가 사용자에게 결과를 전송하고 다음 Fetch Call이 올 때까지 대기한 시간을 더한 값이다. 11번 발생하는 동안 8.43초를 대기했으므로 사용자가 느낀 총 소요시간 8.71초의 대부분을 이 값이 차지한 것을 알 수 있다. 오라클 서버 입장에서는 할 일 없이 대기한 시간이고, App와 Network 구간에서 소모된 시간이다.

커넥션을 맺은 상태에서 쿼리와 쿼리 수행 사이 thinking time이 긴 애플리케이션이나, 서버로부터 데이터를 Fetch하고 클라이언트 내부적으로 많은 연산을 수행한 후에 다음 Fetch Call을 날리는 배치 프로그램에서 특히 이 값이 크게 나타난다. *SQL*Net message to client* 도 Idle 이벤트에 속한다. 클라이언트에게 메시지를 보냈는데, 클라이언트가 너무 바쁘거나 네트워크 부하 때문에 메시지를 잘 받았다는 신호가 정해진 시간보다 늦게 도착하는 경우다.

위 리포트에서 오라클이 일하는 Elapsed time 동안 발생한 대기 이벤트로는 *db file sequential read* 와 *SQL*Net more data to client* 두 개가 있다. *db file sequential read* 는 Single Block Read 방식으로 디스크 블록을 읽을 때 발생하는 대기 이벤트다.

*SQL*Net more data to client* 는 클라이언트에게 전송할 데이터가 남았는데 네트워크 부하 때문에 바로 전송하지 못할 때 발생하는 대기 이벤트다. 5장에서 설명하겠지만 오라클 서버는 내부적으로 SDU(Session Data Unit) 단위로 패킷을 나누어 전송한다. 하나의 SDU 단위 패킷을 전송했는데 잘 받았다는 신호가 정해진 시간보다 늦게 도착하면 대기가 발생하는데, 그때 발생하는 대기 이벤트가 *SQL*Net more data to client* 이다.

Elapsed time은 Response 시점에서 Call 시점을 차감해서 구한다고 했다. 반면, CPU time과 Wait time은 각 발생구간의 시간을 더해서 구하며, 내부 타이머의 측정 단위 때문에 반올림된 수치를 사용한다. 따라서 이론적으로는 Elapsed time이 CPU time과 Wait time을 더한 값이지만 둘 간에는 항상 오차가 발생한다. 심지어 CPU time이 Elapsed time을 초과하기도 한다.

지금까지 Call 통계에서의 CPU time과 Elapsed time이 어떻게 다른지, 그리고 10046 이벤트 트레이스 레벨을 8로 설정했을 때 나오는 이벤트 발생 현황에 대한 분석 방법을 비교적 자세히 설명하였다.

10046 이벤트 트레이스 다음으로 자주 사용하게 되는 것이 10053 트레이스다. 이는 실행계획을 생성하는 CBO의 의사결정 과정을 추적하는 것을 가능케 하며, 이를 통해 옵티마이저가 이상한 돌출 행동을 보이는 원인을 찾아낼 수 있는 경우가 종종 있다.

(2) 다른 세션에 트레이스 걸기

성능 문제가 발생한 튜닝 대상 SQL 목록을 이미 확보했다면 앞에서처럼 자신의 세션에 트레이스를 걸어 문제 SQL의 트레이스 정보를 수집해 분석을 진행하면 된다. 하지만 아직 튜닝 대상 SQL이 수집되지 않은 상황이라면 커넥션 Pool에 놓인 세션 또는 시스템 레벨로 트레이스를 걸어 SQL 수행 정보를 수집해야 한다. 또는 특정 세션에서 심한 성능 부하를 일으키고 있다면 이미 수행 중인 그 세션에 트레이스를 걸어야 하는데, 그럴 때 사용할 수 있는 방법들이 제공되며 버전에 따라 따르다.

오라클 9i에서 Serial 번호가 3인 145번 세션에 레벨 12로 10046 이벤트 트레이스를 걸려면 아래와 같이 하면 된다.

```
SQL> exec dbms_system.set_ev(145, 3, 10046, 12, '');
```

트레이스를 해제할 때는 레벨을 0으로 설정하면 된다.

```
SQL> exec dbms_system.set_ev(145, 3, 10046, 0, '');
```

dbms_system.set_sql_trace_in_session 프로시저를 이용하는 방법도 있으나 오라클은 이것을 사용하지 말 것을 권고하고 있다.

```
dbms_system.set_sql_trace_in_session(sid=>145,serial#=>3,sql_trace=>true);
```

오라클 10g 이후부터는 dbms_monitor 패키지를 사용하면 된다.

```
SQL> begin
  2    dbms_monitor.session_trace_enable (
  3        session_id => 145
  4      , serial_num => 3
  5      , waits => TRUE
  6      , binds => TRUE);
  7  end;
  8  /
```

트레이스를 해제할 때는 session_trace_disable 프로시저를 사용한다.

```
SQL> begin
  2    dbms_monitor.session_trace_disable (
  3       session_id => 145
  4     , serial_num => 3 );
  5  end;
  6  /
```

문제가 발생한 세션에 트레이스를 걸 때, 버전에 상관없이 오래 전부터 사용하던 명령어는 oradebug다. 자세한 사용법은 'oradebug help'를 입력하면 나오지만 기본 사용 패턴만 간단히 예시하면 아래와 같다. 트레이스를 설정하고자 하는 세션의 OSPID를 먼저 확인해야 하는데, 확인된 값이 3796이라고 가정하다.

```
sys@ORA10G> oradebug setospid 3796
명령문을 처리했습니다.
sys@ORA10G> oradebug unlimit   -- 트레이스 파일의 크기 제한을 없앰
명령문을 처리했습니다.
sys@ORA10G> oradebug event 10046 trace name context forever, level 8
명령문을 처리했습니다.
```

이제 해당 세션에 레벨 8로 트레이스가 설정되었다. 트레이스 파일명을 확인하려면 아래처럼 하면 된다.

```
sys@ORA10G> oradebug tracefile_name
d:\oracle\admin\ora10g\udump\ora10g_ora_3796.trc
```

트레이스를 해제하는 방법은 아래와 같다.

```
sys@ORA10G> oradebug event 10046 trace name context off
명령문을 처리했습니다.
sys@ORA10G> oradebug close_trace
명령문을 처리했습니다.
```

시스템 레벨로 전체 세션에 트레이스를 거는 방법은 아래와 같다.

```
SQL> alter system set sql_trace = true;
SQL> alter system set sql_trace = false;
```

물론, 시스템 전체에 트레이스를 걸 때도 10046 이벤트 트레이스를 이용하면 레벨 설정을 할 수 있지만 심각한 부하를 일으키므로 사용할 일이 없다. 부득이한 경우, 짧은 시간 동안만 걸었다가 해제하는 용도로 사용해야 한다.

(3) Service, Module, Action 단위로 트레이스 걸기

최근 개발된 n-Tier 구조의 애플리케이션은 WAS에서 DB와 미리 맺어놓은 커넥션 Pool에서 세션을 할당 받기 때문에 특정 프로그램 모듈이 어떤 세션에서 실행될지 알 수 없고, 한 모듈 내에서 여러 SQL을 수행할 때 각각 다른 세션을 통해 처리될 수도 있다. 이런 환경에서 성능 문제가 발생한 특정 모듈이나 SQL에 대해서만 트레이스를 거는 것은 매우 어려운 작업이다. WAS에서 맺은 세션에만 트레이스를 건 후에 트레이스 파일을 모두 뒤져 해당 모듈에서 수행한 SQL을 찾아내야 하므로 번거롭고 시간도 많이 소요된다. 게다가 커넥션 Pool에 유지되는 세션 개수는 동적으로 늘었다 줄었다 하는데 트레이

스를 설정한 후에 새로 맺어진 세션에 대해서는 속수무책이다. 하는 수 없이 DB 트리거를 이용해 로그온 시점에 트레이스가 걸리도록 하거나 시스템 레벨로 전체 트레이스를 걸어야만 한다.

하지만 10g부터 service, module, action별로 트레이스를 설정하고 해제할 수 있는 dbms_monitor 패키지가 소개되면서 위와 같은 불편함이 모두 사라졌다.

```
SQL> show parameter service_name

NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------
service_names                        string      eCRM

SQL> select sid, service_name, module, action
  2  from   v$session
  3  where  service_name <> 'SYS$BACKGROUND' ;

   SID SERVICE_NAME      MODULE                    ACTION
------ ----------------- ------------------------- --------------------
   135 SYS$USERS         SQL*Plus
   138 eCRM              SQL*Plus
   140 eCRM              PL/SQL Developer          Main session
   144 eCRM              TOAD 8.5.3.2
   145 eCRM              Orange for ORACLE DBA     3.1.5 (Build:5)

5 개의 행이 선택되었습니다.
```

위 쿼리 결과를 보면, 현재 접속해 있는 시스템의 service_name은 eCRM이고, 백그라운드 프로세스를 제외하면 5개 세션이 접속해 있다.

service_name이 eCRM인 세션에 모두 트레이스를 걸려면 아래처럼 하면 된다. 현재 접속해 있는 4개 세션 뿐 아니라 앞으로 새로 커넥션을 맺는 세션 중에서도 service_name이 eCRM이면 자동으로 트레이스가 설정된다.

```
SQL> begin
  2    dbms_monitor.serv_mod_act_trace_enable (
  3          service_name => 'eCRM'   --> 대소문자 구분하므로 주의 !!
  4        , module_name => dbms_monitor.all_modules
  5        , action_name => dbms_monitor.all_actions
  6        , waits => true
  7        , binds => true
  8    );
```