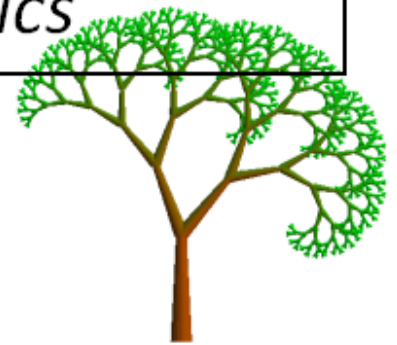




Query Tuning Foundations

Execution Plan

RowSource Statistics



Muriel Fry - Version V1.4 - Sept 2009

- ***Execution Plan***
 - ***Annexe - Row Sources Statistics***



Execution Plan

- The execution plan of a SQL statement is composed of small building blocks called “row sources.”
- The combination of row sources for a statement is called the execution plan.
- By using parent-child relationships, the explain plan can be displayed in a tree (Text or Graphical)

ORACLE®

Execution Plan

An execution plan is the output of the optimizer and is presented to the execution engine for implementation. It instructs the execution engine as to the operations it needs to perform to retrieve the data required by a query most efficiently. The implementation of these operations is down to the execution engine to implement.

When a statement is executed, the optimizer performs many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of steps that are used to run a statement is called an *execution plan*.

An execution plan includes an access method for each table that the statement accesses and an ordering of the tables (the join order). The optimizer also uses different methods to combine the rows from multiple tables (join method). The steps of the execution plan are not performed in the order in which they are numbered. there is parent – child relationship between steps.

<Note.235530.1> Recommended Methods for Obtaining a Formatted Explain Plan

SQLs : Non-Recursive & Recursive

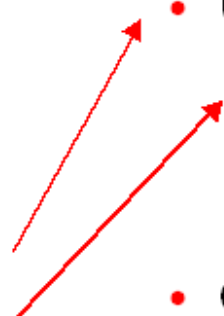
- User SQL
- User SQL
- User SQL
- Commit



- User SQL
 - Internal SQL
 - Internal SQL
- User SQL
- User SQL
 - Internal SQL
 - Internal SQL
 - Internal SQL
- Internal SQL
- Commit

Non-Recursive = (depth = 0)

Recursive = (depth > 0)

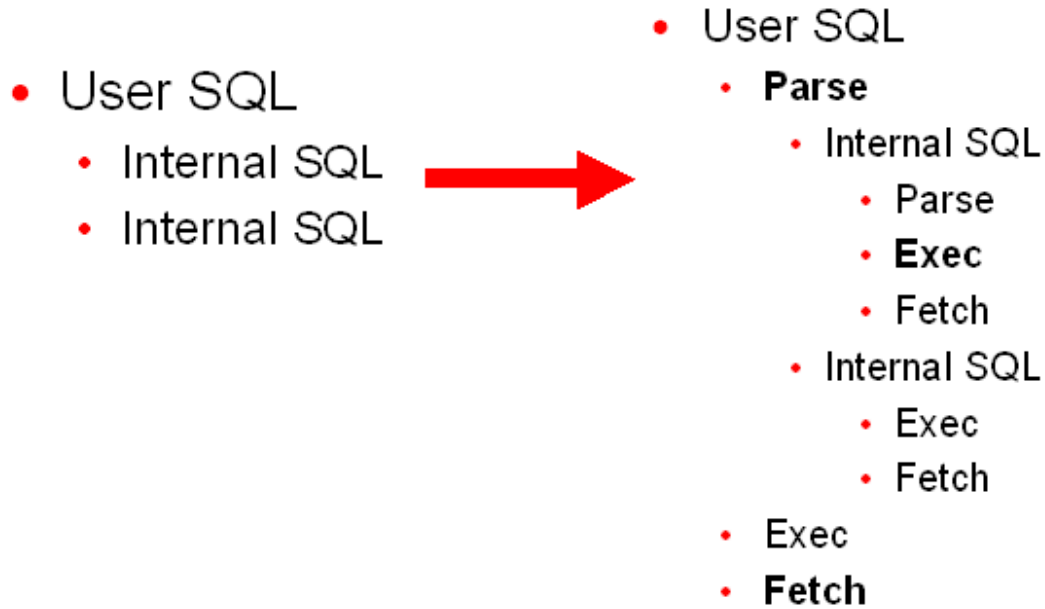


ORACLE

SQLs : Non-Recursive & Recursive



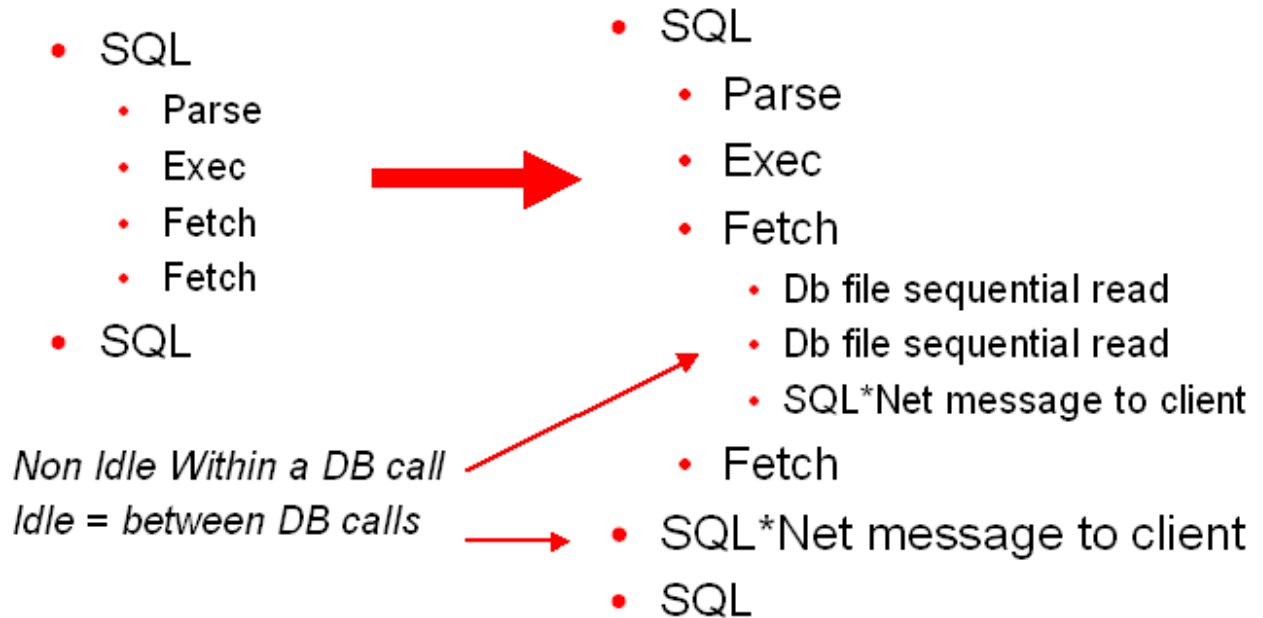
SQLs : Database Calls



ORACLE

SQLs : Database Calls

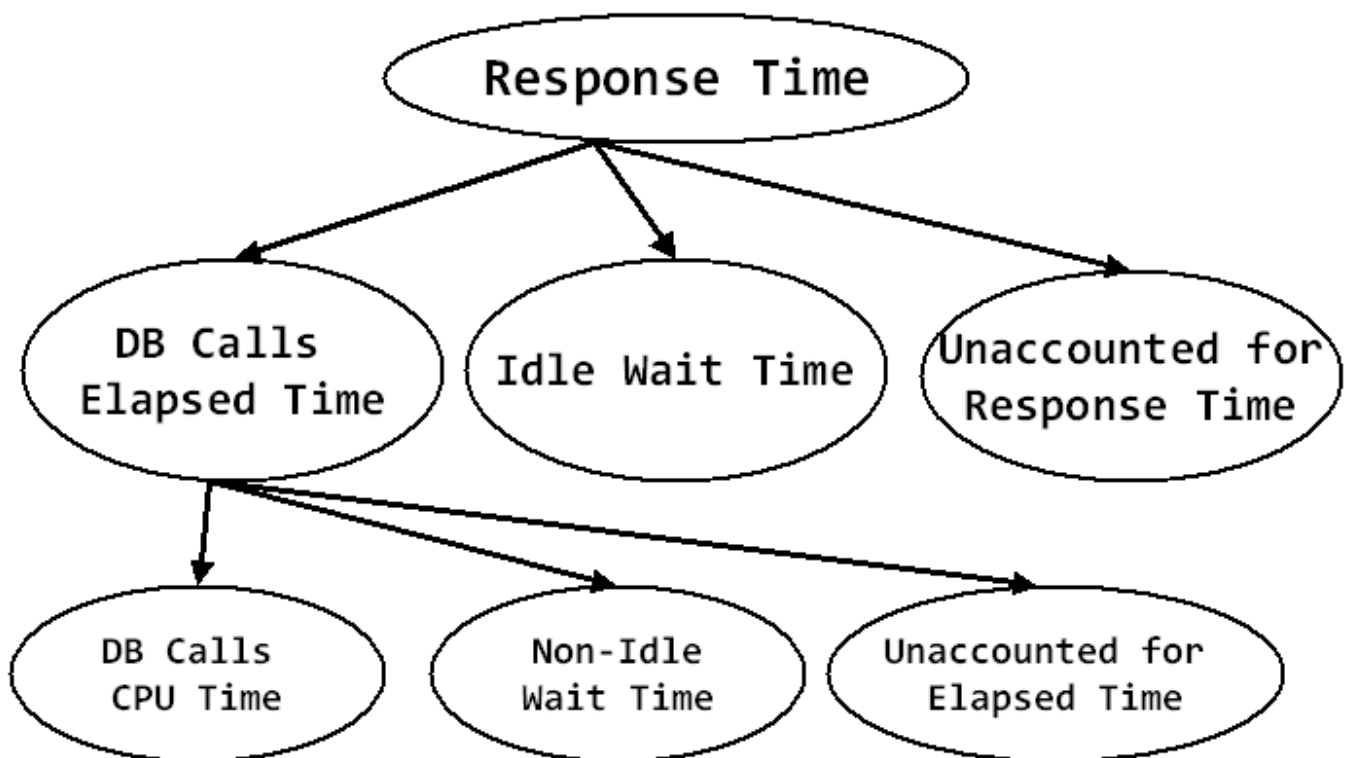
SQLs : Waits Events



ORACLE

SQLs : Wait Events

Response Time Components



ORACLE

Response Time Components



Response Time Components

- Elapsed Time (from DB Calls)
 - CPU Time (from DB Calls)
 - Non-Idle Wait Time (within DB Calls)
 - Elapsed Time Unaccounted-for
- Idle Wait Time (between DB Calls)
- Response Time Unaccounted-for

ORACLE

Response Time Components

DB Call

Database kernel operation, such as "Parse", "Execute", "Fetch", "Unmap" and "Sort Unmap".

CPU Time

Amount of CPU time consumed by one db call, or a set of calls.

Wait Event

Sequence of kernel instructions that consume wall-clock time.

Non-Idle Wait Event

Wait event that originates within a db call, for example "db file sequential read".

Idle Wait Event

Wait event that originates between db calls, for example "SQL*Net message from client".

Non-Idle Wait Time

Wall-clock time or duration of a non-idle wait event.

Idle Wait Time

Wall-clock time or duration of an idle wait event.

Unaccounted-for Time

Under-counted (+) or over-counted (-) time difference between wall-clock time and that recorded in a trace file.

There are several valid reasons for this unaccounted-for time. Refer to literature for further explanation.

Ignore this time slice if it accounts for less than a small threshold (like 10% of total wall-clock time).

Elapsed Time

Wall-clock time of a db call or a set of calls. It includes CPU and non-idle wait times.

Elapsed Time = "CPU" + "Non-Idle Wait" + "Elapsed Unaccounted-for" times.

Response Time

Wall-clock time for a traced process. It is also referred as user time.

Response time has been measured using timestamps of first and last db calls found in trace.

It includes elapsed time and idle wait times. It can be analyzed slicing it into its components in several ways.

Response Time = "End of last db Call" - "Start of first db Call".

Response Time = "Elapsed" + "Idle Wait" + "Response Unaccounted-for" times.

Response Time = "CPU" + "Non-Idle Wait" + "Elapsed Unaccounted-for" + "Idle Wait" + "Response Unaccounted-for" times.

Response Time = "CPU" + "Non-Idle Wait" + "Idle Wait" + "Unaccounted-for" times.

Response Time = "CPU" + "Wait" + "Unaccounted-for" times.

Response Time Accounted-for

Response Time Accounted-for = "Elapsed" + "Idle Wait" times.

Buffer Gets in Consistent Read Mode

Oracle buffers reads from the buffer cache, usually associated with queries.

Buffer Gets in Current Mode

Oracle buffers reads from the buffer cache, usually associated with updates.

Logical IO

Buffer gets from buffer cache in either mode (consistent or current). LIOs are CPU intensive.

PIOs counts are included in LIOs counts.

Operating System Buffer Gets

Oracle blocks obtained from the OS. They are also referred as Physical IOs. PIOs are Non-Idle Wait intensive.



Zoom RAW: Trace (01) 10046 (Entête / Pied)

Dump file c:\oracle\product\10.2.0\admin\fry\udump\fry_ora_3460.trc

Oracle Database 10g Enterprise Edition Release 10.2.0.4.0 - Production...

Windows XP Version V5.1 Service Pack 3

Windows thread id: 3460, image: ORACLE.EXE (SHAD)

*** 2009-08-21 10:41:34.250

*** ACTION NAME:() 2009-08-21 10:41:34.234 DBMS_APPLICATION_INFO

*** MODULE NAME:(SQL*Plus) 2009-08-21 10:41:34.234 DBMS_APPLICATION_INFO

*** SERVICE NAME:(SYS\$USERS) 2009-08-21 10:41:34.234

*** SESSION ID:(150.22) 2009-08-21 10:41:34.234 N° session tracée

Vérifier la fin du fichier :

*** DUMP FILE SIZE IS LIMITED TO 10240000 BYTES *** ????

ORACLE

Zoom RAW : trace(01) 10046 – (Entête / Pied)

I believe that the best way to begin the study of Oracle operational data is with a tour of Oracle's *extended SQL trace* output. SQL trace output presents a linear sequential recorded history of what the Oracle kernel does in response to an application's demands upon the database.

The Oracle kernel helpfully emits a *** line into the trace data any time there has been a significant amount of time (tens of seconds) elapsed since the emission of the previously emitted trace line. This feature is helpful because it allows you to resynchronize your understanding of the correct wall clock time over large spans of WAIT lines, which contain approximate elapsed durations (ela), but no internal clock (tim) values. If you want to emit this line yourself to your trace data, you can do so by calling DBMS_SYSTEM.KSDDDT.

Zoom RAW: Trace 10046 (DB calls)

```
PARSING IN CURSOR #1 len=68 dep=0 uid=47 oct=3 lid=47 tim=1045378263 hv=3063997299 ad='1d72e198'
select 'Bonjour, nous sommes le '||sysdate from dept where deptno=10
END OF STMT
PARSE #1:c=0,e=91,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1045378258
BINDS #1:
EXEC #1:c=0,e=154,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1045379118
WAIT #1: nam='SQL*Net message to client' ela= 4 driver id=1111838976 #bytes=1 p3=0 obj#=-1 tim=1045379259
FETCH #1:c=0,e=44,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=1,tim=1045379488
WAIT #1: nam='SQL*Net message from client' ela= 247 driver id=1111838976 #bytes=1 p3=0 tim=1045379900
FETCH #1:c=0,e=3,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,tim=1045380059
WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1111838976 #bytes=1 p3=0 obj#=-1 tim=1045380194
WAIT #1: nam='SQL*Net message from client' ela= 175987 driver id=1111838976 #bytes=1 p3=0 tim=1047140216
STAT #1 id=1 cnt=1 pid=0 pos=1 obj=47187 op='INDEX UNIQUE SCAN PK_DEPT (cr=1 pr=0 pw=0 time=26 us)'
XCTEND r1bk=0, rd_only=1
```

ORACLE

Zoom RAW : Trace 10046 (DB Calls)

A *database call* is a subroutine in the Oracle kernel. If SQL tracing is active when a database call completes, then the Oracle kernel emits a database call line upon completion of that database call.

PARSE, EXEC, and FETCH calls are the most common types of database call.

Each database call line contains statistics (see next slide)

Note that the Oracle kernel does not emit a database call line into the trace file until the action has *completed*. Thus, an extraordinarily long database operation might cause the Oracle kernel to work for several hours without emitting *anything* to the trace file. Poorly optimized SQL can produce EXEC calls (for updates or deletes) or FETCH calls (for selects) that consume CPU capacity for several *days* at a time.



Detail : PARSE, EXEC, FETCH

PARSE #1: c=0, e=91, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, tim=1045378258

EXEC #1: c=0, e=154, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, tim=1045379118

FETCH #1: c=0, e=44, p=0, cr=1, cu=0, mis=0, r=1, dep=0, og=1, tim=1045379488

c= 0 CPU time in microseconds, rounded to nearest 1/100 of a second
e= 154 Elapsed time in microseconds
p=0 Physical reads
cr=0 consistent reads
cu=0 current mode gets
mis=0 Library Cache misses. If 1, means we just had a hard parse.
r=0 rows processed, i.e., returned
dep=0 recursive depth
og=1 Optimizer mode: 1= All_Rows; 2= First_Rows, 3=Rule, 4= Choose.
tim= 1045379118 The ubiquitous timestamp, in microseconds. Time represents the time at the end of the call.

ORACLE

Detail : PARSE, EXEC, FETCH

Zoom RAW : Trace 10046 (Others - 39817.1)

```
PARSING IN CURSOR #1 len=68 dep=0 uid=47 oct=3 lid=47 tim=1045378263 hv=3063997299 ad='1d72e198'
select 'Bonjour, nous sommes le '||sysdate from dept where deptno=10
END OF STMT
PARSE #1:c=0,e=91,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1045378258
BINDS #1:
EXEC #1:c=0,e=154,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1045379118
WAIT #1: nam='SQL*Net message to client' ela= 4 driver id=1111838976 #bytes=1 p3=0 obj#=-1 tim=1045379259
FETCH #1:c=0,e=44,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=1,tim=1045379488
WAIT #1: nam='SQL*Net message from client' ela= 247 driver id=1111838976 #bytes=1 p3=0 tim=1045379900
FETCH #1:c=0,e=3,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=0,tim=1045380059
WAIT #1: nam='SQL*Net message to client' ela= 3 driver id=1111838976 #bytes=1 p3=0 obj#=-1 tim=1045380194
WAIT #1: nam='SQL*Net message from client' ela= 175987 driver id=1111838976 #bytes=1 p3=0 tim=1047140216
STAT #1 id=1 cnt=1 pid=0 pos=1 obj=47187 op='INDEX UNIQUE SCAN PK_DEPT (cr=1 pr=0 pw=0 time=26 us)'
XCTEND rlbk=0, rd_only=1
```

ORACLE

Zoom RAW : Trace 10046 (Others – 39817.1)

Other database call types (for example, ERROR, UNMAP, and SORT UNMAP) are explained in Oracle *MetaLink* note 39817.1.

The PARSING IN CURSOR line itself contains information about cursor #ID. Text between the PARSING IN CURSOR line and the corresponding END OF STMT line is the cursor's SQL text. The Oracle kernel usually emits this section at the conclusion of a parse call, just before the kernel emits a cursor's PARSE line. However, if tracing was not active when the parse call completed, the kernel will usually emit near the beginning of the trace data (just before the completion of the first traced database call, but potentially after one or more WAIT lines),

An Oracle *wait event* is a sequence of Oracle kernel instructions that is wrapped with special timing instrumentation. If level-8 or level-12 SQL tracing is active when a wait event completes, then the Oracle kernel emits a WAIT line upon completion of that event.

Each WAIT line contains statistics about work executed during the event (see later)

Note that WAIT lines appear in the trace data *before* the database call that motivated them. This occurs because the Oracle kernel emits lines into the trace file as events complete. Thus, if a fetch call requires three OS read calls, the three waits for the

read calls will appear in the trace file before Oracle emits the information about the completed fetch call.

If level-4 or level-12 SQL tracing is active when the Oracle kernel binds values to placeholders in an application's SQL text, the kernel emits a BINDS section (see later).

If SQL tracing is active when a cursor is closed, then the Oracle kernel emits one STAT line for each row source operation in the cursor's execution plan.

If a trace file does not contain the STAT lines you were hoping to find, it is because tracing was deactivated before the cursor closed. The STAT lines will of course be absent any time you trace a well-designed persistent service that neither terminates nor closes its cursors more than once every several weeks.

Each STAT line contains the following statistics about the cursor's execution plan:

- **id** : The unique ID of the row source operation within the STAT line set.
- **cnt** : Number of rows returned by this row source operation.
- **pid** : ID of this operation's parent operation.
- **pos** : The best we can determine, an arbitrary number. It might seem that this value might define the "position" of a row source operation within a set of operations belonging to its parent.
- **obj** : Object ID of the row source operation, if the operation executes upon a "base object." When a row source operation does not access a base object then obj=0.
- **op** : The name of the row source operation.
- **cr** : Number of consistent-mode reads.
- **pr** : Number of Oracle blocks read with OS read calls.
- **pw** : Number of Oracle blocks written with OS read calls.
- **time** : The elapsed duration, expressed in microseconds (us).

The statistics for a parent row source operation include a roll-up of the statistics for its children.

If SQL tracing is active when a commit or rollback occurs, then the Oracle kernel emits an XCTEND line upon completion of the call. For example:

Each XCTEND line contains the following statistics about work executed during the commit or rollback:

- **rlbk** : True (1) if and only if the transaction was rolled back.
- **rd_only** : True (1) if and only if the transaction changed no data in the database.

Notice that the XCTEND marker has no cursor ID reference. This is because there is a one-to-many relationship between a transaction and the cursors that participate in the transaction.

Detail : PARSING IN CURSOR

- Appears when a statement is not already printed in the trace when a PARSE, EXEC, FETCH operation is about to be printed

```
PARSING IN CURSOR #1 len=68 dep=0 uid=47 oct=3 lid=47 tim=1045378263  
hv=3063997299 ad='1d72e198'  
select 'Bonjour, nous sommes le '||sysdate from dept where deptno=10  
END OF STMT
```

CURSOR #1 Gets reused within this session. Enables association of events with each cursor.

len=68 Length of the SQL text

dep=0 Depth.

oct=3 Oracle Command Type. Type of SQL operation. 3= SELECT,
see V\$SESSION in the Database Reference

lid=47 User ID of the user who first parsed the statement

tim= 1045378263 microsecond time since some arbitrary value. Not clock or date time.

hv= 3063997299 Hash value of SQL string. Used to get hash bucket for Library Cache. Not
necessarily unique, unless combined with address. Repeatable value across
platforms for a given Oracle version.

ad= '1d72e198' Address of this object's handle in the Library Cache.

Combined with hash value, this gives an absolute unique identifier of this object
in the Library Cache. Established dynamically when cursor is first created.

ORACLE

Detail : PARSING IN CURSOR #

Most of the time you will see "PARSING IN CURSOR" just before a PARSE operation. But, don't be surprised if you see it at other times when you have a trace that was started during a call that was in progress.

It does NOT mean that the statement was re-parsed, just that Oracle was about to print information related to PARSE/EXEC/FETCH and discovered it had not already printed the PARSING IN CURSOR line.



Detail : Binds if level 4 or 12 chosen

BINDS #1: Bind#0

```
oacdtty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbf=08228e44 bln=22 avl=04 flg=05
value=47186
```

oacdtty =2	Data type
mxl=22(22)	Maximum length, private max length (may not always apply)
mal=00	Total # of array elements
scl=00	Scale, for datetime and interval types
pre=00	Precision, also for datetime and interval types
oacflg=13	Bitmask for various things (internal uses)
fl2=1	Extension of oacflg
size=24	Size and offset identify location of value for this bind variable
kxsbbbf=0555ded8	UGA buffer pointer
bln=22	Buffer length
avl=02	Value length
flg=05	Bitmask, (internal uses)
value=2	Actual value of bind variable

ORACLE

Detail : Binds if Level 4 or 12 is chosen

Bind#0

```
oacdtty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
kxsbbbf=08228e44 bln=22 avl=04 flg=05
value=47186
```

A BINDS section contains one or more bind subsections, one for each variable being bound.

The number following the word bind indicates the ordinal position, beginning at 0, of the bind variable within the SQL text.

- **dtty** : The external data type of the value supplied by the application [Oracle OCI (1999)]. Oracle publishes two sets of data types: *internal* and *external*.
 - o •The internal data type definitions reveal how the Oracle kernel stores its data on the host operating system.

- o The external data type definitions reveal how the Oracle kernel interfaces with application SQL.
- **avl** : The length, in bytes, of the bind value.
- **value** : The value that is bound into the statement execution. The Oracle kernel sometimes truncates values that it emits into the trace file. check if avl value is larger than the length of the value field.
- o fl2 : correspond à oacfl2 utile pour savoir si une Bind est Safe ou non dans le cadre du cursor_sharing
<reference uacdef.h> :
 - @ #define UACFBLTR 0x00000100 /* Bind was generated by LiTeRal replacement */
 - @ #define UACFUNSL 0x00000200 /* **UNSafe** Literal */
 - @ #define UACFNDDL 0x00000400 /* Non-DaTa LiteRal */



Detail : Wait Events

- Wait events always occur **BEFORE** the action it relates to is reported
- If you see the event printed, it already happened
- Always look for the following call with the same cursor # as the wait event

```
WAIT #1: nam='SQL*Net message from client' ela= 247 driver id=1111838976 #bytes=1 p3=0 tim=1045379900
```

nam='SQL*Net...'	Name of event, see v\$event_name
ela= 247	Elapsed time, in microseconds
p1=1111838976	P2, P3 are specific to the event
p2=1	Refer to WebIV for event of interest
p3=0	

ORACLE

Detail : Wait Events

- **nam** : The name assigned by an Oracle kernel developer to reveal which part of the Oracle kernel code is responsible for this portion of your response time.
- **ela** : The elapsed duration of the named event's execution. Oracle9i expresses ela in microseconds (1 μ s = 0.000001 seconds). Prior kernel versions express ela in centiseconds (1 cs = 0.01 seconds).
- **p1, p2, p3** : The meanings of these parameters vary by nam. A complete catalog of parameter descriptions for each event type is available by running the following SQL: select name, parameter1, parameter2, parameter3 from v\$event_name order by name

Zoom RAW : Trace 10046 (Wait Events – Idle vs NonIdle)

```
PARSING IN CURSOR #2 len=57 dep=0 uid=47 oct=3 lid=47 tim=2769298867 hv=4052346060 ad='1d6c14d0'
select 'Hi, I come from '||deptno||' '||dname from dept
END OF STMT
PARSE #2: c=0, e=5683, p=0, cr=3, cu=0, mis=1, r=0, dep=0, og=1, tim=2769298863
BINDS #2:
EXEC #2: c=0, e=211, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=1, tim=2769299946
WAIT #2: nam='SQL*Net message to client' ela= 11 driver id=1111838976 #bytes=1 p3=0 obj#=47186 tim=2769300088
WAIT #2: nam='db file sequential read' ela= 61034 file#=4 block#=298163 blocks=1 obj#=47186 tim=2769361787
WAIT #2: nam='db file scattered read' ela= 1517 file#=4 block#=298164 blocks=5 obj#=47186 tim=2769363749
FETCH #2: c=0, e=63720, p=6, cr=3, cu=0, mis=0, r=1, dep=0, og=1, tim=2769364000
WAIT #2: nam='SQL*Net message from client' ela= 379 driver id=1111838976 #bytes=1 p3=0 obj#=47186 tim=2769364551
WAIT #2: nam='SQL*Net message to client' ela= 3 driver id=1111838976 #bytes=1 p3=0 obj#=47186 tim=2769365035
FETCH #2: c=0, e=209, p=0, cr=5, cu=0, mis=0, r=3, dep=0, og=1, tim=2769365222
WAIT #2: nam='SQL*Net message from client' ela= 5951350 driver id=1111838976 #bytes=1 p3=0 obj#=47186
tim=2775316803
STAT #2 id=1 cnt=4 pid=0 pos=1 obj=47186 op='TABLE ACCESS FULL DEPT (cr=8 pr=6 pn=0 time=63692 us)'
XCTEEND r1blk=0, rd_only=1
```

ORACLE®

Zoom RAW : Trace 10046 (Wait Events – Idle vs Non Idle)

The fundamental relationship of Oracle time statistics within a single database call: the total elapsed duration (e) approximately equals the total CPU time for the call (c) plus the sum of the durations of its wait events (Σelap)

You should attribute each WAIT #n duration to the first database call for cursor #n that *follows* the WAIT line in the trace file.

For events executed *within* database calls, the reason for forward attribution is easy to understand. Because lines are written to the trace file as their corresponding actions complete, the wait events executed by a given database call appear in the trace stream *before* the call's trace file line.

Ligne[7] : At the conclusion of the EXEC call, the kernel shipped a result back to the client program (that is, the SQL*Plus program). The write to the socket consumed 11 µs of elapsed time.

Ligne[10] : The FETCH statistics show an elapsed duration of 63 µs to return one row (r=1), which required three reads of the database buffer cache, all in consistent mode (cr=3) and none in current mode (cu=0) and 6 blocks from disk

Ligne[8] et Ligne[9] : WAIT event to read these 6 blocks from disk

Ligne[13] : The next database call recorded in the trace file is another fetch, which took place after a 379-µs read from the SQL*Net socket and 3µs write to the socket. The fetch returned 3 rows and consumed only 209 µs of elapsed time.

After shipping back the fetch result to the client, the kernel sat idle awaiting its next request. It came after 5.95 sec (exit), cursor had been closed at that time with its statistics, and Read Only transaction finished.

RELIRE RAPIDEMENT LE LIVRE DE MILLSAP (BUREAU)

Virtually every trace file you'll ever analyze will have some mismatch between actual response time and the amount of time that the Oracle kernel accounts for in its trace file

Activation/ Déactivation par une session tierce (des éléments sont manquants dans la hiérarchie : DB call (CPU + Somme(Waits)du DB Call) et SQL/SQL récursif (Temps du récursif inclus dans le temps du SQL)

Response Time Unaccounted-for is basically the difference between user elapsed time (as per begin/end timestamps on trace) and the total elapsed time (including idle waits). In other words, that is that time that elapsed but there is no record of it on the trace. (Par exemple temps dans la runqueue)

If the trace was started or stopped in the middle of a call, how much unaccounted for time is there in the trace?

Zoom : Trace 10046 (Recursive)

```
PARSING IN CURSOR #1 len=210 dep=1 uid=0 oct=3 lid=0 tim=2769294703 hv=864012087 ad='21268d5c'
select /*+ rule */ bucket_cnt, row_cnt, cache_cnt, null_cnt, timestamp#, sample_size, minimum, maximum,
distcnt, lowval, hival, density, col#, spare1, spare2, avgcln from hist_head$ where obj#=1 and intcol#=:2
END OF STMT

PARSE #1:c=0,e=28,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=3,tim=2769294695
BINDS #1:
  Bind#0
    oacdty=02 mxl=22(22) mxlc=00 mal=00 scli=00 pre=00 fl2=0001 ... value=47186
  Bind#1 ...

EXEC #1:c=0,e=1708,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=3,tim=2769297372
FETCH #1:c=0,e=122,p=0,cr=3,cu=0,mis=0,r=1,dep=1,og=3,tim=2769297623
STAT #1 id=1 cnt=1 pid=0 pos=1 obj=255 op='TABLE ACCESS BY INDEX ROWID HIST_HEAD$ (cr=3 pr=0 time=127 us)'
STAT #1 id=2 cnt=1 pid=1 pos=1 obj=257 op='INDEX RANGE SCAN I_HH_OBJ#_INTCOL# (cr=2 pr=0 pw=0 time=77 us)'
=====
PARSING IN CURSOR #2 len=57 dep=0 uid=47 oct=3 lid=47 tim=2769298867 hv=4052346060 ad='1d6c14d0'
select 'Hi, I come from '||deptno||' '||dname from dept
END OF STMT

PARSE #2:c=0,e=5683,p=0,cr=3,cu=0,mis=1,r=0,dep=0,og=1,tim=2769298863
```

ORACLE

Zoom : Trace 10046 (Recursive)

The term *recursive* denotes the Oracle kernel's execution of database calls within the context of other database calls.

Activities that inspire recursive SQL include execution of DDL statements, execution of PL/SQL blocks with DML statements within them, database call actions with triggers on them, and all sorts of routine application DML statements that motivate data dictionary access. Any database call that can execute another database call can motivate recursive SQL.

A trace file excerpt containing evidence of recursive SQL. The three cursor #1 actions at dep=1 are recursive children of the dep=0 parse action upon cursor #2

The rule for determining the recursive relationships among database calls is simple:

A database call with dep= $n + 1$ is the recursive child of the first subsequent dep= n database call listed in the SQL trace data stream. The Oracle kernel can emit trace data for a database call only after the action has completed. (The kernel cannot compute, for example, the call's elapsed time until after the call has completed.)

A database call's c, e, p, cr, and cu statistics contain an aggregation of the resources consumed by the database call itself and its entire recursive progeny.



10046 Tracing: Potential Problems

- Due to Collection Methods
 - Trace activated during long running call
 - Compute unaccounted time for first call in trace
 - Trace deactivated during long running call
 - Compute missing time at end of trace
 - Trace activated/deactivated during a recursive call
 - Look for missing child calls associated with a parent; or, missing parents associated with a child
- Due to unaccounted time

ORACLE

10046 Tracing : Potential Problems

Unaccounted-for Time

Process not executing (waiting in OS ready queue)

Visiting a non-instrumented event

Others (usually small)

Measurement instrumentation overhead (time record keeping)

Quantization error (CPU measured in CS, not in micro-secs)

Double-counting CPU (negative unaccounted-for time)

Counted twice: by DB Call and by OS Call from DB Call



10046 Tracing: Potential Problems

- Due to Architecture
 - Shared Server and Connection Pooling
- Set DBMS_SESSION.SET_IDENTIFIER at the beginning of your session (via logon trigger, OCI or JDBC)
- Instrument your code using DBMS_APPLICATION_INFO
- Turn on Statistics and tracing using DBMS_MONITOR
- Use TRCSESS if required to consolidate trace files
- Use RAW trace file, tkprof or another parser to format the output file

ORACLE

10046 Tracing : Potential Problems

Lorsque l'application se connecte à la db, elle s'identifie via un "client_id" ; c'est via ce client_id que l'on va pouvoir tracer le run d'une application même si la connection à la db se fait au travers d'un pool de connection .

1) activer la trace sur base d'un identifiant "client" et non sur base de la session physique (sid,serial#)

```
SQL> exec dbms_monitor.client_id_trace_enable(client_id=>'TEST');
```

ceci vaut pour toutes les sessions existantes et futures

2) Initialisation du client_id

```
exec dbms_session.set_identifier('TEST');
```

ceci doit se faire dans le code application à chaque connection --> cela permet de suivre un utilisateur réel au sein d'un connection pool

3) Consolider les différents fichiers de trace, et les formater via tkprof

```
>C:\...\udump> trcsess output="tracefile.trc" clientid='TEST'  
>C:\...\udump> tkprof TEST.trc output=TEST.tkp
```

4) Disable de la trace si plus nécessaire

```
SQL> exec dbms_monitor.client_id_trace_disable(client_id=>'TEST');
```



DBMS_XPLAN summary

```
select t1.*
from emp t2 , emp t1
where t1.empno = t2.empno
      and t1.sal > 500
      and t2.empno between 5000 and 9000
```

ORACLE

DBMS_XPLAN

```
select t1.*
from emp t2 , emp t1
where t1.empno = t2.empno
      and t1.sal > 500
      and t2.empno between 5000 and 9000
```

The first section of output is as follows:

SQL_ID 392agcpnsb5qy, child number 0

```
select t1.* from emp t2 , emp t1 where t1.empno = t2.empno      and t1.sal > 500
and t2.empno between 5000 and 9000
```

- The **sql_id** identifies the parent cursor. This information is available only when the output is generated by the functions `display_cursor` and `display_awr`.
- The **child number**, along with the `sql_id`, identifies the child cursor. This information is available only when the output is generated by the function `display_cursor`.
- The text of the SQL statement is available only when the output is generated by the functions

display_cursor and display_awr.

SQL_ID is **endianness** sensitive. So if 2 platforms have different endianness then SQL_ID will be different for the same SQL text.

The second section shows the hash value of the execution plan and, in a table, the execution plan itself.

Plan hash value: 866206949

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		14	574	2 (0)	00:00:01
1	NESTED LOOPS		14	574	2 (0)	00:00:01
* 2	TABLE ACCESS BY INDEX ROWID	EMP	14	518	2 (0)	00:00:01
* 3	INDEX RANGE SCAN	PK_EMP	14		1 (0)	00:00:01
* 4	INDEX UNIQUE SCAN	PK_EMP	1	4	0 (0)	00:00:01

In the table, estimations and execution statistics for each operation are provided. The number of columns in the table depends directly on the amount of available information. For example, information about partitioning, parallel processing, or execution statistics is shown only when available.

Column Description

Basics (Always Available)

Id The identifier of each operation (line) in the execution plan. If the number is prefixed by an asterisk, it means that predicate information for that line is available later.

Operation The operation to be executed. This is also known as the row source operation.

Name The object on which the operation is executed.

Query Optimizer Estimations

Rows (E-Rows) The estimated number of rows returned by the operation.

Bytes (E-Bytes) The estimated amount of data returned by the operation.

TempSpc The estimated amount of temporary space used by the operation.

Cost (%CPU) The estimated cost of the operation. The percentage of CPU cost is given in parentheses. Note that this value is cumulated through the execution plan. In other words, the cost of parent operations contains the cost of their child operations.

Time The estimated amount of time needed to execute the operation (HH:MM:SS).

Partitioning

Pstart The number of the first partition to be accessed. Set to either KEY, KEY(I),KEY(MC), KEY(OR), or KEY(SQ) if unknown at parse time.

Pstop The number of the last partition to be accessed. Set to either KEY, KEY(I), KEY(MC), KEY(OR), or KEY(SQ) if unknown at parse time.

Parallel and Distributed Processing

Inst For distributed processing, the name of the database link used by the operation.

TQ For parallel processing, the table queue used for the communication between slaves.

IN-OUT The relationship between parallel or distributed operations.

PQ Distrib For parallel processing, the distribution used by producers to send data to consumers.

Runtime Statistics*

Starts The number of times a specific operation was executed.

A-Rows The actual number of rows returned by the operation.

A-Time The actual amount of time spent executing the operation (HH:MM:SS.FF).

Buffers The number of logical read operations performed during the execution.

Reads The number of physical reads performed during the execution.

Writes The number of physical writes performed during the execution.

OMem The estimated amount of memory needed for an optimal execution.

1Mem The estimated amount of memory needed for one-pass execution.

O/1/M The number of times the execution was performed in optimal/one-pass/multipass mode.

Used-Mem The amount of memory used by the operation during the last execution.

Used-Tmp The amount of temporary space used by the operation during the last execution. This value must be multiplied by 1,024 to be consistent with the other memory utilization columns (for example, 32K means 32MB).

Max-Tmp The maximum amount of temporary space used by the operation. This value has to be multiplied by 1,024 to be consistent with the other memory utilization columns (for

example, 32K means 32MB).

- Available only when execution statistics are enabled.

The third section shows the query block names and the object aliases:

Query Block Name / Object Alias (identified by operation id):

```
-----  
1 - SEL$1  
2 - SEL$1 / T1@SEL$1  
3 - SEL$1 / T1@SEL$1  
4 - SEL$1 / T2@SEL$1
```

For each operation in the execution plan, you see which query block it is related to and, optionally, on which object it is executed. This section is available as of Oracle Database 10g. So here when we have same table (EMP) used in join, we are able to determine that Operation 4 is relative to T2 and operation 2 is relative to T1.

The fourth section is available only as of Oracle Database 10g Release 2. It shows the set of hints that are necessary to force that particular execution plan. This set of hints is called outline.

Outline Data

```
/*+  
    BEGIN_OUTLINE_DATA  
    IGNORE_OPTIM_EMBEDDED_HINTS  
    OPTIMIZER_FEATURES_ENABLE('10.2.0.4')  
    ALL_ROWS  
    OUTLINE_LEAF(@"SEL$1")  
    INDEX_RS_ASC(@"SEL$1" "T1"@"SEL$1" ("EMP"."EMPNO"))  
    INDEX(@"SEL$1" "T2"@"SEL$1" ("EMP"."EMPNO"))  
    LEADING(@"SEL$1" "T1"@"SEL$1" "T2"@"SEL$1")  
    USE_NL(@"SEL$1" "T2"@"SEL$1")  
    END_OUTLINE_DATA  
*/
```

The fifth section shows which predicates are applied. For each of them, it is shown where (line) and how (access or filter) they are applied.

Predicate Information (identified by operation id):

```
2 - filter("T1"."SAL">500)  
3 - access("T1"."EMPNO">=5000 AND "T1"."EMPNO"<=9000)  
4 - access("T1"."EMPNO"="T2"."EMPNO")  
    filter(("T2"."EMPNO">=5000 AND "T2"."EMPNO"<=9000))
```

While an access predicate is used to locate rows by taking advantage of an efficient access structure (for example, a hash table in memory when hash join or an index like for operation 3), a filter predicate is applied only after the rows have already been extracted from the structure storing them.

The following section shows which columns are returned as output when each operation is executed. It is available as of Oracle Database 10g only.

Column Projection Information (identified by operation id):

```
-----

1 - "T1"."EMPNO"[NUMBER,22], "T1"."ENAME"[VARCHAR2,10],
   "T1"."JOB"[VARCHAR2,9], "T1"."MGR"[NUMBER,22], "T1"."HIREDATE"[DATE,7],
   "T1"."SAL"[NUMBER,22], "T1"."COMM"[NUMBER,22], "T1"."DEPTNO"[NUMBER,22]

2 - "T1"."EMPNO"[NUMBER,22], "T1"."ENAME"[VARCHAR2,10],
   "T1"."JOB"[VARCHAR2,9], "T1"."MGR"[NUMBER,22], "T1"."HIREDATE"[DATE,7],
   "T1"."SAL"[NUMBER,22], "T1"."COMM"[NUMBER,22], "T1"."DEPTNO"[NUMBER,22]

3 - "T1".ROWID[ROWID,10], "T1"."EMPNO"[NUMBER,22]
```

Finally, there is a section providing notes and warnings about the optimization phase, the environment, or the SQL statement itself:

Note

```
-----
- Warning: basic plan statistics not available. These are only collected when:
  * hint 'gather_plan_statistics' is used for the statement or
  * parameter 'statistics_level' is set to 'ALL', at session or system level
```

Query with hints to see Outline DATA generated

```
SQL> select /*+ ordered use_hash(t1) full(t1) */ t2.* from emp t2 , emp
t1
2   where t1.empno = t2.empno
3      and t1.sal > 500
4      and t2.empno between 5000 and 9000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

14 rows selected.

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ADVANCED LAST ALLSTATS'));
```

```
SQL_ID 1sw1hbmff14xjh, child number 0
```

```
-----
select /*+ ordered use_hash(t1) full(t1) */ t2.* from emp t2 , emp t1
```

where t1.empno = t2.empno and t1.sal > 500 and t2empno between 5000 and 9000

Plan hash value: 3750662990

Id	Operation	Name	Str	E-R	E-By	Cst(%CPU)	A-R	Buf	OMem	lMem	Usd-Mem	
* 1	HASH JOIN			1	14	630	6 (17)	14	10	762K	762K	1028K(0)
2	TABLE ACCESS BY INDEX ROWID	EMP		1	14	518	2 (0)	14	2			
* 3	INDEX RANGE SCAN	PK_EMP		1	14		1 (0)	14	1			
* 4	TABLE ACCESS FULL	EMP		1	14	112	3 (0)	14	8			

Query Block Name / Object Alias (identified by operation id):

```

1 - SEL$1
2 - SEL$1 / T2@SEL$1
3 - SEL$1 / T2@SEL$1
4 - SEL$1 / T1@SEL$1

```

Outline Data

```

/*+
  BEGIN_OUTLINE_DATA
  IGNORE_OPTIM_EMBEDDED_HINTS
  OPTIMIZER_FEATURES_ENABLE('10.2.0.4')
  ALL_ROWS
  OUTLINE_LEAF(@"SEL$1")
  INDEX_RS_ASC(@"SEL$1" "T2"@"SEL$1" ("EMP"."EMPNO"))
  FULL(@"SEL$1" "T1"@"SEL$1")
  LEADING(@"SEL$1" "T2"@"SEL$1" "T1"@"SEL$1")
  USE_HASH(@"SEL$1" "T1"@"SEL$1")
  END_OUTLINE_DATA
*/

```

Predicate Information (identified by operation id):

```

1 - access(T1.EMPNO=T2.EMPNO)
3 - access(T2.EMPNO>=5000 AND T2.EMPNO<=9000)
4 - filter((T1.SAL>500 AND T1.EMPNO>=5000 AND T1.EMPNO<=9000))

```

Column Projection Information (identified by operation id):

```

1 - (#keys=1) T2.EMPNO[NUMBER,22], T2.DEPTNO[NUMBER,22], T2.ENAME[VARCHAR2,10],
  T2.JOB[VARCHAR2,9], T2.MGR[NUMBER,22], T2.HIREDATE[DATE,7], T2.SAL[NUMBER,22],
  T2.COMM[NUMBER,22]
2 - T2.EMPNO[NUMBER,22], T2.ENAME[VARCHAR2,10], T2.JOB[VARCHAR2,9],
  T2.MGR[NUMBER,22], T2.HIREDATE[DATE,7], T2.SAL[NUMBER,22], T2.COMM[NUMBER,22],
  T2.DEPTNO[NUMBER,22]
3 - T2.ROWID[ROWID,10], T2.EMPNO[NUMBER,22]
4 - T1.EMPNO[NUMBER,22]

```

- The hash join at line 1 uses the predicate T1.EMPNO=T2.EMPNO to join the two tables. In other words, the access predicate might show a join condition as well. In this specific case, the access predicate is used to specify that the hash table in memory containing the data of table T1 will be probed with the values returned by accessing table T2
- The index scan at line 3 accesses the index PK_EMP to look up the column empno of the table T2. In this case, the access predicate shows on which key the lookup is performed.
- At line 4, all rows in table T1 are read through a full scan. Then, when the rows have been extracted from the blocks, the predicate ((T1.SAL>500 AND T1.EMPNO>=5000 AND T1.EMPNO<=9000)) is applied to filter them.

