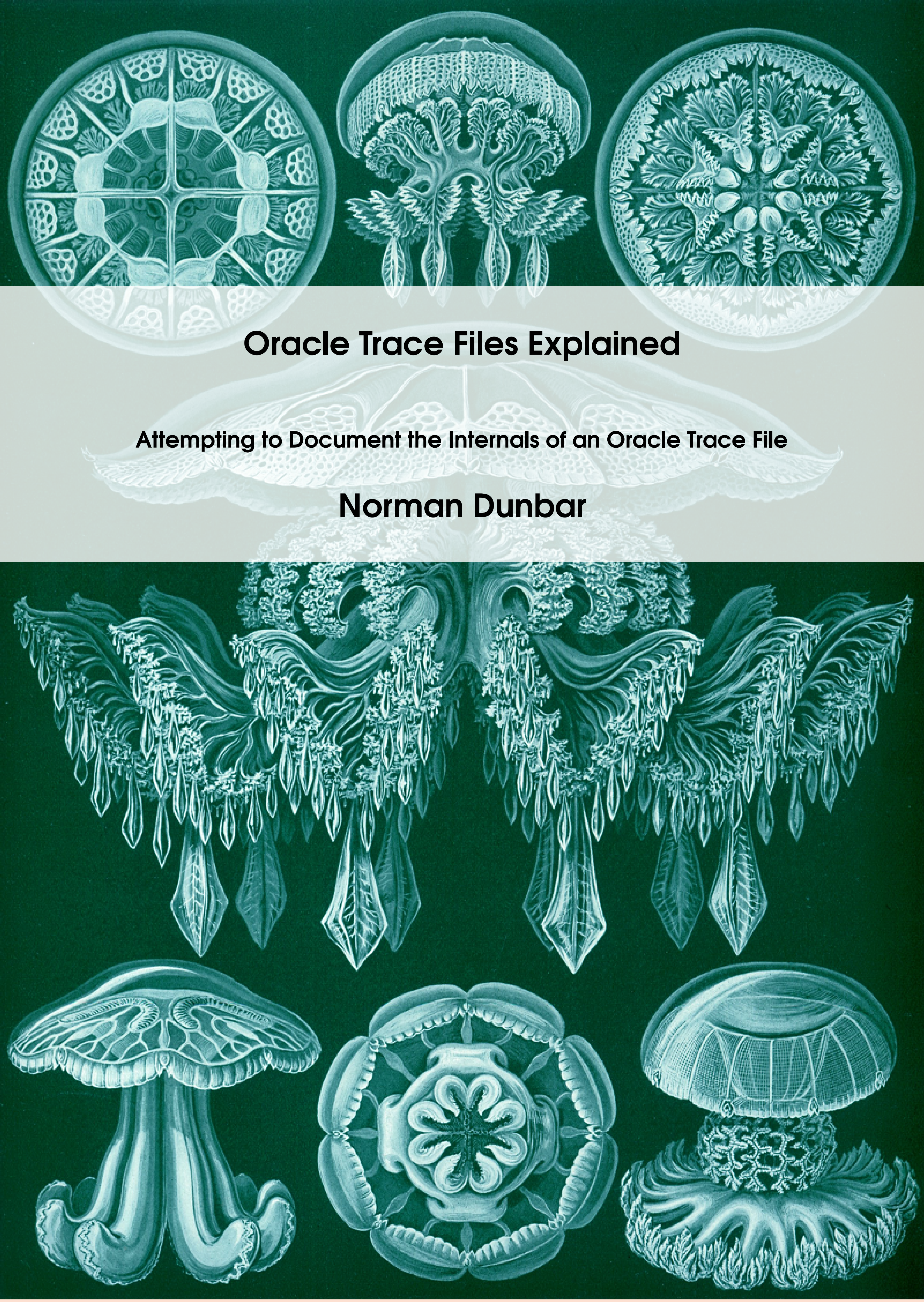


Oracle Trace Files Explained

Attempting to Document the Internals of an Oracle Trace File

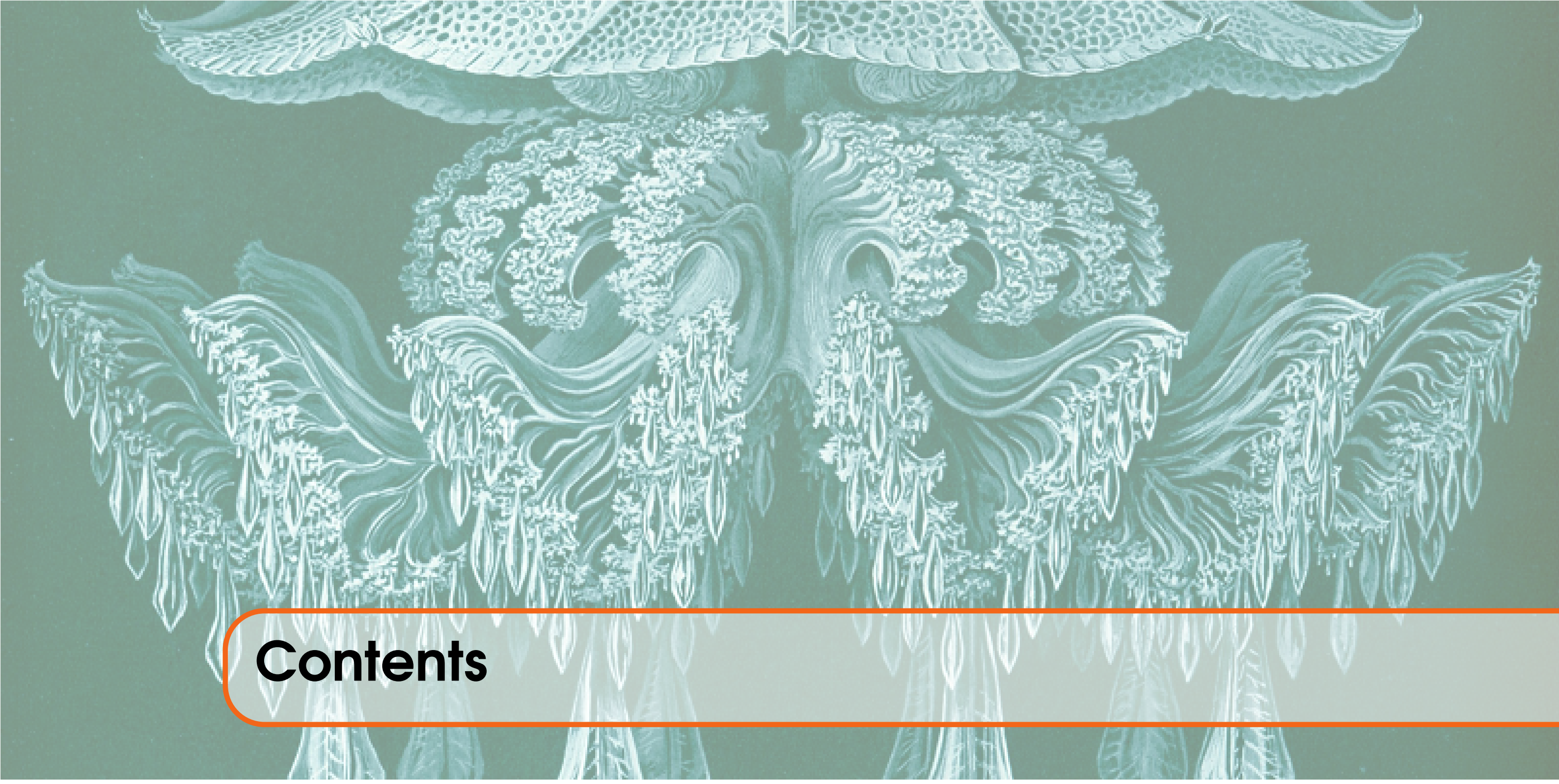
Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

The latest version of the eBook can be downloaded from the book's github repository.

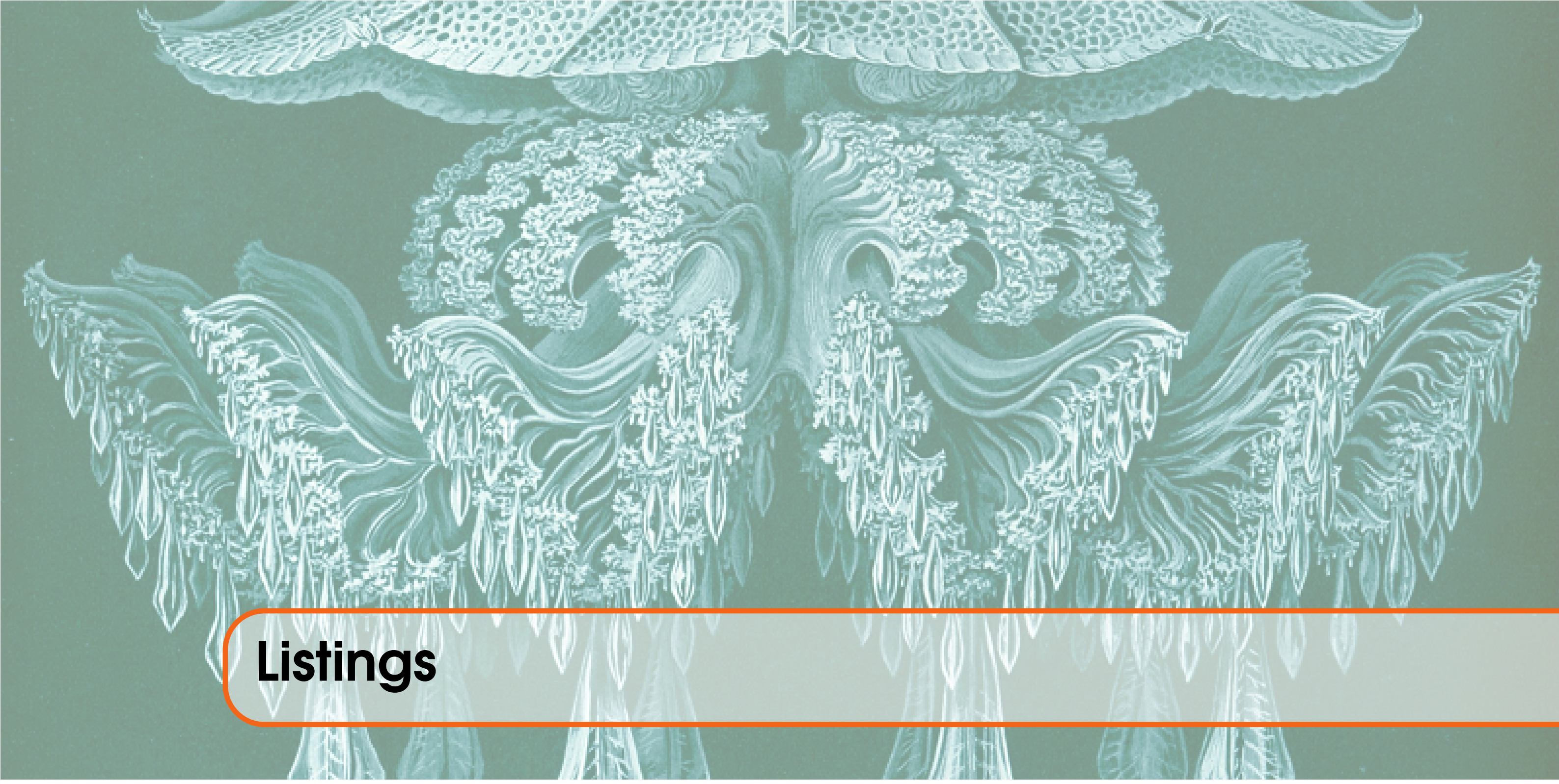
First printing, July 2017

This pdf document was created on *21/6/2018* at *10:12:29*.



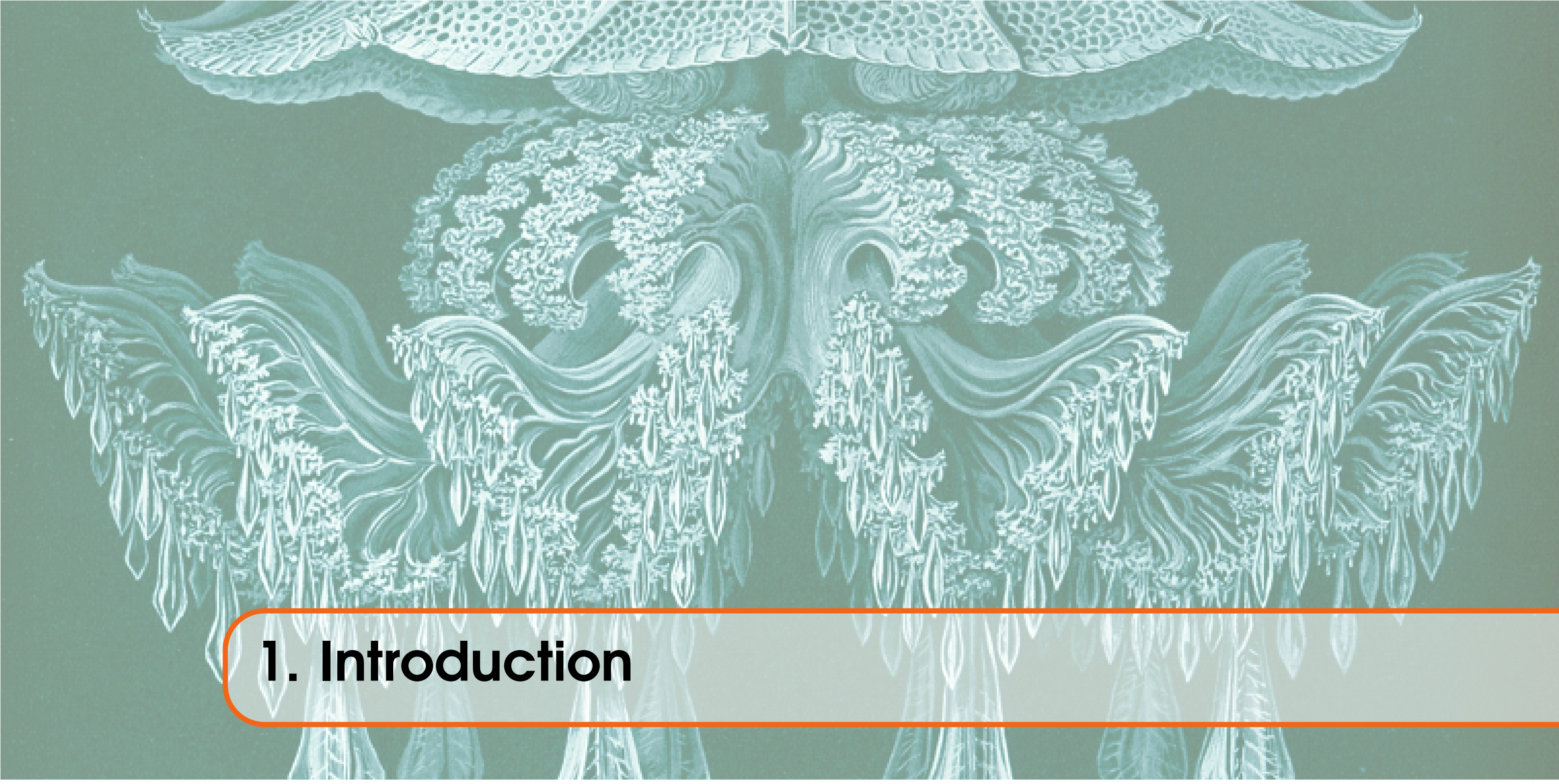
Contents

List of Tables



Listings

Oracle Trace Files



1. Introduction

Oracle trace files are not greatly documented. This document is an attempt to do so. It is *not* official in any way and is based on a good few years of reading these files to help diagnose various database problems.

A trace file is really the best way to delve into what Oracle is doing, or to discover why something is taking so long - it shows you exactly what happened during the period that the session was being traced.

Even better, when you extract an Explain Plan from a trace file, it is showing you exactly how Oracle retrieved the data from the tables, and exactly where the time was spend in doing so. Running an `Explain Plan for ...` statement in *SQL*Plus*, *Toad* etc tells you what Oracle *might* do. The two are not always the same.

The trace files used (and abused) in this eBook were all self contained, and extracted from a dedicated session on the database. If you are using shared sessions, then you have the unfortunate problem of having bits of your session trace spread over any number of different trace files. Not fun.

From Oracle 10g onwards, a utility has been supplied, named *trcsess*. This allows you to collect together all the separate files and merge them into one, ready for analysis. Even better, it apparently can also merge Oracle 9i shared server session trace files. Bonus!

Trace File Extracts

In the remainder of this eBook, extracts from trace files will be shown as follows:

```
WAIT #3220341128: nam='db file sequential read' ela= 1023 file#=3 block#=12 blocks=1 obj#=-1 tim=3520817183625
```

You will notice that long lines from the trace file are wrapped in the example above. However, all such wrapped lines are indicated by a $\Longrightarrow$ at the start of the continuation line(s). Sometimes though, the break in the lines can be at an awkward position. Sadly, I can't help this, it's all automagically done and I have no input as to how, exactly, LaTeX decides.

Code Listings

SQL scripts (or any other program listings) will be listed as per the example below. The only difference being line numbers, in case I have to reference a particular point, and syntax highlighting.

```sql
select name, parameter1, parameter2, parameter3
from   v$event_name
where  name = 'db file sequential read';
```

Irregular Updates

This eBook is a work in progress, I am always learning, or trying to. As I update it, the latest version will always appear on GitHub as detailed on the Copyright page above. Also on that page, you will be able to see the date and time that the version of the eBook you are reading, was generated. That's done automagically too, so I don't need to remember to update it - thankfully!

Comments

Seen anything blatantly wrong? Something you think needs a better explanation? Too many blatant plugs? You can either create an issue on GitHub, or, email me with details and I'll do my best to get it fixed.

2. Trace File Gotchas!

Trace files can be full of pitfalls and traps for the unwary. Best take care!

2.1 Things to Beware Of

There are a few things to watch out for in trace files, some of these are mentioned frequently in the text below, but they are also gathered here together, in one place, for your convenience.[1]

2.1.1 When Are Lines Written to the Trace File?

It cannot be said often enough, *a line in the trace file is written when that particular action has completed* and not before. So, if you see a line defining a `PARSE` statement in the trace file, the `PARSE` has completed at that point. The `tim` value gives you the (relative) time that the action completed.

2.1.2 Tim Values

On 11g `tim` values are in microseconds, or, millionths of a second, and are an offset from some epoch, which itself depends on the Operating System in use.

You can get a rough idea of how long a statement took by subtracting the previous `tim` from the current `tim` - the answer is in microseconds, so count back 6 places from the right end of the `tim` value, and shove in a decimal point.

Alternatively, use my `TraceAdjust` utility to convert `tim` values into date and times accurate to the microsecond. (Plug! See Appendix E on page 53 for details.)

[1] Ok, for *my* convenience!

2.1.3 Timestamps

Not all actions in the trace file may be *accurately* timed. This can lead to the various `tim` values getting slightly out of step. To attempt to alleviate this, Oracle will write timestamp lines to the trace file. These are flagged by three asterisks (`***`) followed by the date and time, accurate to microsecond level, possibly in the format `yyyy-mm-dd hh24:mi:ss.u` but this *might* depend on locality etc. The `tim` value immediately following these timestamp lines are adjusted to match the timestamp that preceded it.

You can see an example of a timestamp line in the header for the trace file. An example is in Section 3.1.1 on Page 9.

2.1.4 Cursor Depth

Cursors are parsed, executed etc at a given depth. This can be seen in the `dep=`*n* details in the appropriate lines in the trace file.

If the listed depth is zero, then this is the top level SQL in the trace file, and usually corresponds to your own statements. Statements running in a PL/SQL procedure, function etc, however, *may* be listed at a different depth with the call to the procedure (or whatever) being run at depth zero.

As mentioned above, lines are written to the trace file when an action completes, so you will see (at least in most trace files) the `PARSE`, `EXEC` and `FETCH` and possibly `STAT` and/or `CLOSE` for the recursive SQL statements *after* the `PARSE` but *before* the `EXEC` for your statement. Simply because of the need to execute all those recursive statements in order to facilitate the `EXEC` of yours.

If you see a statement being actioned at dep=n, where $n <> 0$, then it is being actioned recursively for another statement at dep=$n-1$ which *follows later* in the trace file.

2.1.5 Cursor IDs Get Reused

When you see something like `PARSE #1234567890` then the digits after the # are the cursor ID. In the old days, prior to Oracle 9i, cursor IDs were monotonically increasing and were never reused in the same trace file. Once cursor #1 had been closed, you wouldn't see it again.

This is no longer true and a cursor ID *can* be reused once closed. There are two (known to me) variations:

Same SQL Statement Reused by Same CursorID

Some applications write SQL that is parsed far too frequently and may even be parsed each and every time it is executed by the application code. If this is the case, the trace files *may* show the following steps

- `CLOSE` of the current cursor.
- *Possibly* a `PARSE` for the new statement. This will not be present if the previous `CLOSE` resulted in the cursor being cached.
- And so on.

If the same statement is reparsed and nothing else has been parsed since the `CLOSE`, then it is possible that the expected `PARSING IN CURSOR` line, showing the SQL text, may be missing from the trace file.

If the same statement has been parsed three or more times already, Oracle may cache the cursor rather than hard `CLOSE`ing it as this is a performance improvement if the SQL has to be `PARSE`d and EXECuted again. In this case, there will be no PARSE line in the trace file for the statement - simply because it was not actually PARSEd, merely retrieved from the cache already PARSEd.

Different SQL using Same Cursor ID

If a statement's cursor is closed and another *different* statement is parsed, it *can* obtain the same Cursor ID as the recently closed cursor. In this case, you will see the following:

- CLOSE of the current cursor.
- PARSING IN CURSOR and the SQL text of the new statement.
- PARSE for the new statement.
- And so on.

The moral to this tale is simple, you cannot be sure that all the EXECs for a given Cursor ID actually mean that the exact same SQL statement has been executed - you could be seeing lots of different SQL statements simply reusing the same Cursor ID.

2.1.6 What About Child Cursors?

Following on from the above, the same SQL statement *can* (and does) have the ability to be executed under any number of *different* Cursor IDs! How is this possible?

If your statement uses bind variables, then given a recent version of Oracle, something called *Bind Variable Peeking* may be in use to ensure that a statement has the most efficient plan of execution regardless of the bind variable values. In the good old days, whatever bind value got in first determined the execution plan for every other EXEC of that statement, even if it made the execution plan completely hopeless for most values that followed.

I think - because I have not checked - that this means that certain statements and bind combinations will cause a number of separate child cursors, and thus, different Cursor IDs in the trace file, for the same statement. A recent trace file I examined had 15 different Cursor IDs for the same statement.

I knew they were all the same as the PARSING IN CURSOR line has the sqlid at the end, and although the Cursor IDs were different, the SQL Text and such like were all the same and so was the sqlid so it *had* to be the same SQL statement, just with numerous child cursors. Beware of this when looking in trace files!

If you know the sqlid then you can count and find any and all child cursors, and their IDs in the trace file with a swift grep command.

The following command will count the number of child cursors in the trace file for the SQL statement with sqlid 0um91dczrf666:

```
grep -i "0um91dczrf666" TraceFile.trc | wc -l
```

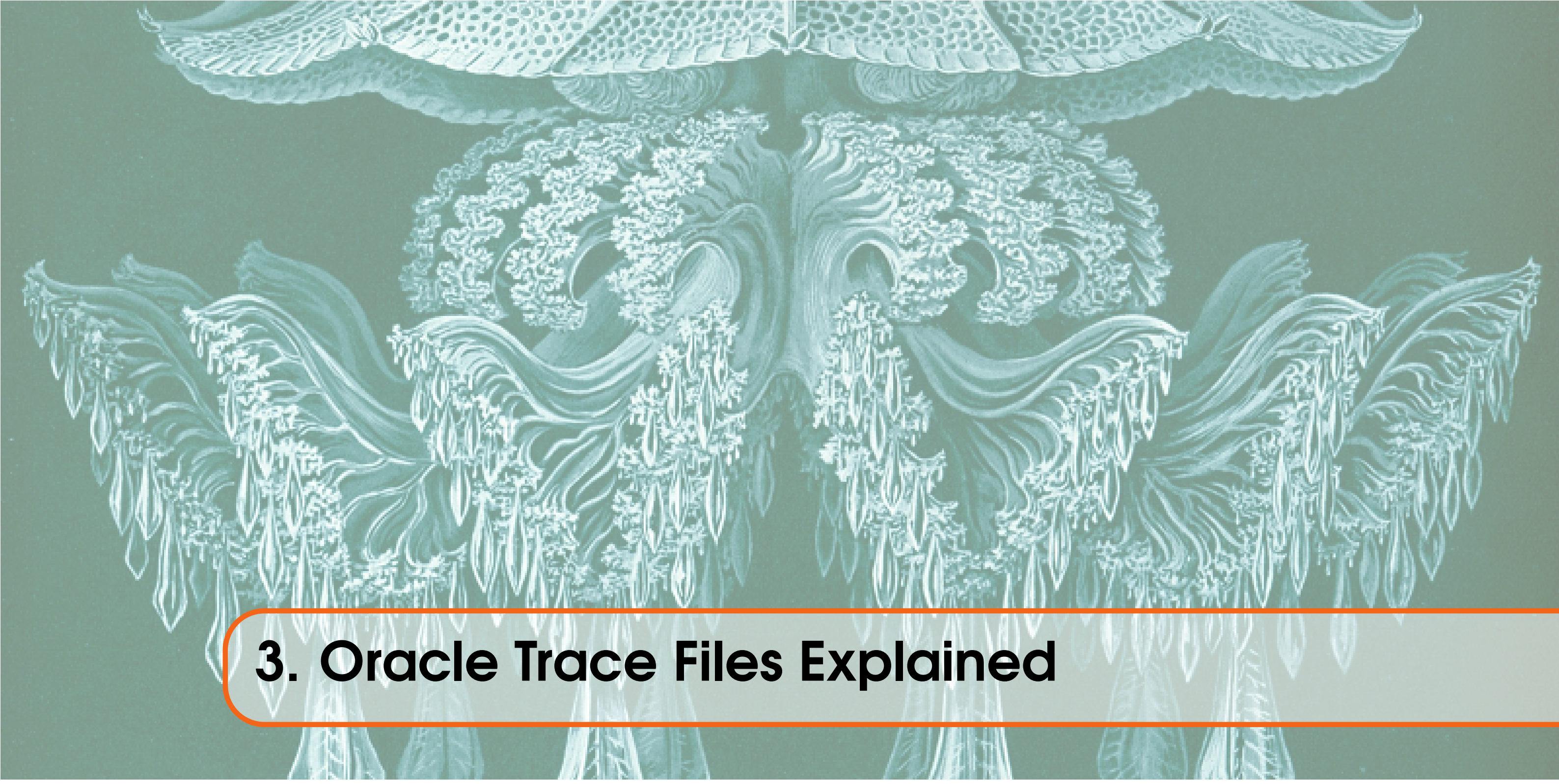
Listing 2.1: Counting Child Cursors in a Trace File

The following command will extract, in the order of appearance in the trace file, all the child cursors for the same statement as above:

```
grep -i "0um91dczrf666" TraceFile.trc | cut -c 20-39
```

Listing 2.2: Extracting Child Cursors from a Trace File

This was an HP-UX trace file, so the digits after the # in the Cursor ID were from position 20 to 39 in the output from grep. Other Operating Systems may differ, so test on yours first, just in case.



3. Oracle Trace Files Explained

3.1 Trace File Sections

The trace file is made up of two main sections, the header and the trace details.

3.1.1 The Header

The header is the top of the file and consists of a few lines of text giving details of where the trace file came from, which server (operating System) it was created on, various details about the server and the database and so on.

The following is an example of a trace file created on a Windows server, running Oracle 11.2.0.4. Server, database and other potentially sensitive information has been obfuscated to protect the guilty, me!

```
Trace file C:\ORACLEDATABASE\diag\rdbms\cfg\cfg\trace\orcl_ora_27680_FREESPACE.trc
Oracle Database 11g Enterprise Edition Release 11.2.0.4.0 - 64bit Production
Windows NT Version V6.2
CPU                 : 8 - type 8664, 8 Physical Cores
Process Affinity    : 0x0x0000000000000000
Memory (Avail/Total): Ph:29917M/57343M, Ph+PgF:24634M/65535M
Instance name: orcl
Redo thread mounted by this instance: 1
Oracle process number: 373
Windows thread id: 27680, image: ORACLE.EXE (SHAD)


*** 2017-06-27 13:56:36.872
*** SESSION ID:(1017.1085) 2017-06-27 13:56:36.872
*** CLIENT ID:() 2017-06-27 13:56:36.872
*** SERVICE NAME:(ORCLSRV) 2017-06-27 13:56:36.872
```

```
*** MODULE NAME:(TOAD background query session) 2017-06-27 13:56:36.872
*** ACTION NAME:() 2017-06-27 13:56:36.872

====================
```

Listing 3.1: Oracle 11g Trace File Header

The last line, consisting of equals signs, is the separator between the header and the following trace details.

In Oracle versions before 10g, the latter chunk of text is not found, the one detailing session, client, module etc, those first appeared at 10g.

Note

Sometimes, you might see a header that has a number of detail lines before the final separator, for example:

```
...

*** 2017-05-08 12:39:13.496
*** SESSION ID:(777.2309) 2017-05-08 12:39:13.496
*** CLIENT ID:() 2017-05-08 12:39:13.496
*** SERVICE NAME:(SYS$USERS) 2017-05-08 12:39:13.496
*** MODULE NAME:(SQL*Plus) 2017-05-08 12:39:13.496
*** ACTION NAME:() 2017-05-08 12:39:13.496

WAIT #523653448: nam='SQL*Net message to client' ela= 1
    ⟹ driver id=1111838976 #bytes=1 p3=0 obj#=14232 tim
    ⟹ =1743670562625
WAIT #523653448: nam='SQL*Net message from client' ela= 1027
    ⟹ driver id=1111838976 #bytes=1 p3=0 obj#=14232 tim
    ⟹ =1743670564245
CLOSE #523653448:c=0,e=7,dep=0,type=1,tim=1743670564280
WAIT #0: nam='SQL*Net more data from client' ela= 12 driver
    ⟹ id=1111838976 #bytes=10 p3=0 obj#=14232 tim
    ⟹ =1743670564312
====================
```

Listing 3.2: Oracle 11g Trace File Header - With Waits etc

In this case, the trace was started after the session had parsed a query, and was in the process of executing it (`EXEC` and, if necessary `FETCH`) when the trace started. Oracle has not shown the `PARSING IN CURSOR` line(s) for the query in question, which is a shame.

Normally, you can ignore the detail lines above the header separator, however, you sometimes see numerous `WAIT` lines above the separator, in which case, there could be something to investigate. This is more likely to occur when you start tracing a session that is taking a huge amount of time, and is already under way when you begin the trace.

3.1.2 Trace Details

The majority of the trace file consists of the full trace details for the session that was traced. There are numerous lines of text here, each different, each with their own fields of one kind or another.

These are explained in the following sections.

Note

Remember, when Oracle writes a line to the trace file, it is done at the *end* of the process that was indicated by the written line. For example, if you see a `FETCH` line in the trace file, that was written at the time (`tim`) that the `FETCH` completed. The trace file should be considered a list of things *that have happened* and not *things that are about to happen*.

For example, a recent trace file that I had the pleasure of examining had over 2 million lines. The problem was to determine where a statement was losing over 2,000 seconds of time on its first `EXEC` which it did not have on any of the subsequent `EXEC`s.

The statement was parsed at `dep=0` on line 36,656 but was not executed, again at `dep=0` until line 284,530, as the following `grep` output shows:

```
grep -n "dep=0" tracefile.trc

...

36656:PARSE #1152921504566889382 4:c=0,e=1076,p=0,cr=0,cu=0,mis=1,r=0,
    ⟹ dep=0,og=1,plh=0,tim=26375808573637

284530:EXEC #1152921504566889382 4:c=83870000,e=2048188431,p=1233,cr
    ⟹ =45536,cu=18,mis=1,r=1,dep=0,og=1,plh=0,tim=26377856794348

...
```

Listing 3.3: Oracle 11g Problem Trace - `grep` output

From the above you can see that nothing took place between the `PARSE` and the `EXEC`, so any other SQL executed between those lines was done in order to facilitate the `EXEC` at line 284,530. So that's where the time was lost, in recursive SQL statements, executed in order to carry out the required statement's `EXEC`.

A quick investigation showed that Oracle was gathering dynamic stats against numerous dictionary tables in the SYS schema, so after a quick `DBMS_STATS.GATHER_SCHEMA_STATS` and a `DBMS_STATS.GATHER_DICTIONARY_STATS` the lost time was restored and the problem went away.

We *know* that the time was being lost in the `EXEC` as the `PARSE` of the statement had been written to the trace file at line 36,656 so at that point, the `PARSE` had completed. The `EXEC` did not complete until line 284,530, so everything between those lines was related to the `EXEC` and nothing to do with the `PARSE`.

Also, the *elapsed time* for the `EXEC` shows that the execution of the statement took 2048.188431 seconds (*e=2048188431*).

Timestamp Lines

One line that you should be interested in is this one from the header above:

```
*** 2017-06-27 13:56:36.872
```

Listing 3.4: Time Stamp Line

This is the first timestamp line in the trace file and sets the baseline for all the `tim` fields (these will be explained below) that follow, however, briefly, the `tim` values are in microseconds (millionths of a second) from a specific "epoch" - which depends on the operating system - and there isn't a consistent, operating system independent, method of converting `tim` values from a huge number of microseconds to an actual date and time that humans will understand.

There are usually a few timestamp lines written to the trace file, depending on how long it has been processing for, and these mean that we can, with a bit of fiddling, relate a `tim` value to an actual time on the clock.

Tim Values

Many trace lines hold a `tim` field which holds an offset from some Operating System defined epoch. In trace files up to Oracle 9i, these values were in hundredths of a second. Since 9i, these are now in micro seconds - or millionths of a second.

The *first* `tim` value that you see, following a timestamp line, will be the microsecond equivalent to the date and time in the timestamp line.

Recursive SQL

Your SQL statements are normally executed as top-level statements, but Oracle might need to execute some (a lot!) of recursive SQL statements, in order that your statement can be processed.

If, for example, you drop a user in a database with the `drop user xxx cascade` statement, Oracle goes off and executes hundreds of separate SQL statement to find out what objects the user owns, or has privileges to, and undoes all of those before finally dropping the contents of the user and finally the user itself.

Top-level SQL statements are identified by having a depth of zero. This can be seen in many of the trace file lines as `dep=0` in the various lines of the trace file.

Recursive statements, executed in the background, have a depth greater than zero, and some of these require recursive statements of their own, and so on.

This recursion leads to a foible in the trace file, your statement appears last and all the possibly nested, recursive statements will normally appear first. This is simply because in order for your statement to be executed, the recursive statements have to run to completion *first*.

For example, in a trace file I have open in front of me, the first statement with a `dep=0` occurs at line 709 in the file. Everything prior to that runs at `dep=3`, `dep=2` or `dep=1` and complete before I can see my own SQL statement.

Under normal circumstances, a statement that is parsed (executed etc) at `dep=`n, where $n > 0$, has been called recursively, to facilitate a statement, that will follow in the trace file, that is itself parsed (executed etc) at `dep=`$n-1$.

Waits

`WAIT` lines in a trace file are similar, in that the `WAIT` must complete, and so is written to the trace file, *before* the statement that incurred the wait. For example, a `FETCH` that had to wait for `db file scattered read` events, will appear later in the trace file than the individual `WAIT` lines that the `FETCH` suffered from.

Cursor Ids

Every time you see a '#' followed by a number, you are looking at a cursor ID. In previous versions of Oracle, these were simply an ever increasing number, starting from 1 and increasing by 1 for each new cursor.

In Oracle 11g, the cursor *appears* to be an address in memory[1], and *will be reused* as cursors are closed and new ones opened. You cannot assume, therefore, that a cursor with a specific ID at the end of the trace file, relates to any other lines with that same ID previously written to the trace file, without checking for any intervening `CLOSE` lines with the same ID - that's just how it is now!

Active or Inactive

You might want to look in the `STATUS` column in `V$SESSION` to determine if the session is active - doing *something* - or not. However, you must be aware that the `STATUS` column only gives an active reading when the statement is in a `PARSE`, `EXEC` or `FETCH` phase of execution. If the statement is in a `WAIT`, for example, it will show as inactive - but it is actually still processing the current statement. Something to bear in mind.

3.2 Trace File Details

As mentioned above, the trace file details, which consists of many different entries, is discussed in the following chapter.

[1] But don't quote me on this, I saw it written down somewhere on the Oracle Support web site, but now that I need it, I cannot find it again. Sigh!

4. Trace File Entries

4.1 PARSING IN CURSOR

This is *usually* the first line you will see for a cursor. It shows the full SQL statement between the `PARSING IN CURSOR` line and the `END OF STMT` line. The SQL is displayed exactly as the user (or application) entered it. However, if the statement is invalid, or cannot be parsed, you will not see this, or the `PARSE` line in the trace file, you will only see a `PARSE ERROR` line instead.

This is not the actual `PARSE` for the cursor though, that normally follows on later, usually!

As an example, here is the `PARSING IN CURSOR` line for the SQL query that Toad runs in the background to extract the free space used in the database by various tablespaces, including temporary ones. However, I'm not showing the SQL here:

```
PARSING IN CURSOR #3220341128 len=3081 dep=0 uid=0 oct=3 lid=0 tim
    ⟹ =3520788574727 hv=3219027813 ad='7ffcb6778350' sqlid='7
    ⟹ bwtj5azxwxv5'
```

Listing 4.1: Parsing In Cursor Line

The various fields defined, and their descriptions can be seen in the table below.

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
len	The size, in bytes? Characters? of the SQL statement.
dep	Recursion level. 0 = Top-level, user, SQL.
uid	The user id of the user parsing the statement.
oct	Oracle Command Code of the SQL Statement. (See Appendices.)
lid	The user id of the proxy user, that the session logged in via. Usually the same as the UID.

Table 4.1: Parsing in Cursor - Fields... *continues on next page*

Code	Description
tim	Time, in microseconds, that the details were written out to the trace file.
hv	Hash Value for the statement.
ad	SQLTEXT address - see V$SQLAREA and/or V$SQLTEXT.
sqlid	The SQL ID for the statement.

Table 4.1: Parsing in Cursor - Fields

The uid field references the column USER_ID in the views DBA_USERS, ALL_USERS and USER_USERS.

The lid field also references the column USER_ID in the views DBA_USERS, ALL_USERS and USER_USERS. However, this only has a value when when logged in via a proxy user. This is the USER_ID of the username in the square brackets when you login - connect norman[dunbar]... both, as far as I have seen, the uid and lid values are the same, and both refer to the 'dunbar' user, not the 'norman' user. Given the same example as above, but logged in as a proxy user, we see something like the following:

```
PARSING IN CURSOR #3220341128 len=3081 dep=0 uid=220 oct=3 lid=220 ...
```

Listing 4.2: Parsing In Cursor Line - Proxy User

Both the uid and the lid parameters are the same, 220, and this is the USER_ID for the 'dunbar' user, although I logged in as the 'norman' user - but that logs in as a proxy for the 'dunbar' user, as in connect norman[dunbar].... I have yet to see the uid and lid numbers differ when logged in as a proxy user - so far at least.

As mentioned, the cursor ID field has a value that may (or may not) be an address in memory. However, that's not the same as the ad field, which is (I think) an address in memory for the cursor.

The sqlid field is the same as the SQL_ID column in V$SQL. The hv field is the hash value that Oracle used to determine if this statement was to be found in the cache or not. It matches the SQL_HASH_VALUE (and PREV_HASH_VALUE) column in V$SESSION and also the HASH_VALUE column in V$SQL.

Sometime Oracle will not write these lines to the trace file. If the cursor has been parsed, and subsequently closed, then re-parsed, you will see a PARSE line but not a new PARSING IN CURSOR for the statement. This *appears* to be only on those occasions where the statement is re-parsed *immediately* after being closed. I have not seen this "feature" when the cursor ID was used by a different statement in between.

4.2 PARSE

Normally, after the PARSING IN CURSOR lines, you will see a PARSE line for the same cursor ID. This is not always the case, for example, if you started the trace after the statement had been parsed, Oracle *may* write the PARSING IN CURSOR lines, but not the PARSE, to the trace file.

Other times when the PARSE line will not be seen is when the cursor associated with the SQL statement was previously CLOSEd but Oracle decided not to hard close the cursor but instead cached it for future use. Normally this happens if SESSION_CACHED_CURSORS is non-zero and the SQL statement has already been PARSEd at least three times.

See the type parameter for the CLOSE trace line in Section 4.9 on Page 28 for details of how and when a cursor *might* be cached.

A typical PARSE line will look like this:

```
PARSE #491311368:c=0,e=452,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh
    ⟹ =1388734953,tim=97734887542
```

Listing 4.3: Parse Line

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
p	The number of physical reads (blocks) that were necessary in order to carry out this PARSE.
cr	The number of consistent reads (blocks) that were necessary in order to carry out this PARSE.
cu	The number of current reads (blocks) that were necessary in order to carry out this PARSE.
mis	Whether this statement was found in the cache (0) or not (1). Indicates whether or not a hard parse was required. Seeing a 1 is bad, usually.
r	Rows processed.
dep	Recursion level. 0 = Top-level, user, SQL.
og	Optimiser goal. 1 = ALL_ROWS, 2 = FIRST_ROWS, 3 = RULE, 4 = CHOOSE. Depending on your version of Oracle, you may not see some of the above.
plh	Execution plan hash value.
tim	Time, in microseconds, when the parse for the statement completed. **Not** the time it took.

Table 4.2: Parse - Fields

The plh value is a value that corresponds to the column PLAN_HASH_VALUE in VSQL_PLAN, VSQL_PLAN_STATISTICS_ALL and V$SQLSTATS. (There may be other views where this value appears, depending on the Oracle version in use.)

Block Details

The `p`, `cr` and `cu` statistics are the usual ones for block reads, viz:

- Physical reads occur when Oracle must read a block, from disc, into the buffer cache.
- Consistent reads are the number of blocks of UNDO that Oracle applied to one or more blocks, in order to return the data in those blocks, to the state they were in when the "query" started. The query in this case, would be any or all recursive queries necessary to facilitate the `PARSE`.
- Current Reads are blocks that Oracle used without needing to roll them back to the start of the "query".

The fields for an `PARSE` are the same as those for an `EXEC` and a `FETCH` - see below.

The `r` field is *interesting*. Surely only a `FETCH` would process some rows? I thought so too, and in all the trace files I've come across, I have yet to see (ok, yet to *notice*) a `PARSE` line with anything other than `r=0`. I wonder why Oracle have it as part of the `PARSE`? Maybe, because they use the same format of line for the `PARSE` and `EXEC` they decided just to use the same one, and set the row count for a `PARSE` to zero?[1]

[1] We may never know!

4.3 PARSE ERROR

If a PARSE goes wrong for some reason, the trace file will show something like the following:

```
PARSE ERROR #491311368:len=26 dep=0 uid=755 oct=3 lid=755 tim
    ⟹ =97734836031 err=1031
SELECT LIVE_DEV FROM SITE
```

Listing 4.4: Parse Error Line

There will not normally be a PARSING IN CURSOR or a PARSE shown in the trace file for this statement.

If there was a password in the SQL text that failed to parse, that will be obfuscated and the statement truncated for security reasons.

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
len	The length of the SQL statement in bytes, maybe characters depending on your characterset.
dep	Recursion level. 0 = Top-level, user, SQL.
uid	The user id of the user parsing the statement.
oct	Oracle Command Code of the SQL Statement. (See Appendices.)
lid	The user id of the proxy user, that the session logged in via. Usually the same as the UID. See Section 4.1 on page 15 for more details about proxy logins.
tim	Time, in microseconds, when this parse error was detected.
err	The Oracle error code that caused the parse to fail.

Table 4.3: Parse Error - Fields

In this case, there was an error ORA-01031: insufficient privileges as the user parsing the statement did not have the required privileges to see the table in question.

The uid field references the column USER_ID in the views DBA_USERS, ALL_USERS and USER_USERS.

4.4 EXEC

The EXEC phase of a statement's execution is when Oracle dives into the database to build a result set of the desired data. It need not be the complete result set though, so beware of that.

A typical EXEC will resemble the following:

```
EXEC #3220341128:c=0,e=101,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh
    ⟹ =2215247290,tim=3520788606189
```

Listing 4.5: Exec Line

Bear in mind, however, that the EXEC elapsed times do not necessarily bear any resemblance to the entire time it took for the user to get a response. That response time includes the PARSE, EXEC, all the FETCHes and all the WAITs that were encountered.

The fields in an EXEC are as follows:

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
p	The number of physical reads (blocks) that were necessary in order to carry out this PARSE.
cr	The number of consistent reads (blocks) that were necessary in order to carry out this PARSE.
cu	The number of current reads (blocks) that were necessary in order to carry out this PARSE.
mis	Whether this statement was found in the cache (0) or not (1). Indicates whether or not a hard parse was required. Seeing a 1 is bad, usually.
dep	Recursion level. 0 = Top-level, user, SQL.
r	Rows processed.
og	Optimiser goal. 1 = ALL_ROWS, 2 = FIRST_ROWS, 3 = RULE, 4 = CHOOSE. Depending on your version of Oracle, you may not see some of the above.
plh	Execution plan hash value.
tim	Time, in microseconds, at which this EXEC statement was completed.

Table 4.4: Exec - Fields

The fields for an EXEC are the same as those for a PARSE (and FETCH - see below).

The r field is *interesting*. Surely only a FETCH would process some rows? Not necessarily. Normally, when EXECing a statement, the number of rows processed is indeed zero, but some PL/SQL, for example, returns a row count of 1. In my demonstration trace file I see 4 statements whose EXEC has r=1 and all of them are of the format:

```
begin
    do_something();
```

```
end;
```

Listing 4.6: PL/SQL Exec Example

The “do_something()” call is to DBMS_MONITOR.START_TRACE, DBMS_MONITOR.STOP_TRACE and calls to DBMS_OUTPUT.GET_LINE. These statements have no FETCH calls.

You may wish to refer back to page 18 for details of the p, cr and cu block statistics in the EXEC line.

4.5 FETCH

After processing the EXEC call, and any associated WAITs, Oracle may FETCH some, or all, of the rows returned by the query. A FETCH line may resemble the following:

```
FETCH #3220341128:c=1812500,e=28665431,p=11137,cr=50774,cu=316,mis=0,r
  ⟹ =30,dep=0,og=1,plh=2215247290,tim=3520817271640
```

Listing 4.7: Fetch Line

And anyone paying close attention will notice that the time it took to FETCH 30 rows was a little excessive at 28.66 seconds. In case you are wondering, this is a query against DBA_FREE_SPACE on an 11.2.0.4 database, and it is taking so long as it is hitting the *never to be solved* Bug 19125876 which affects 11.20.4 on AIX on Power Systems. Unfortunately, *this* trace file is from a Windows server. The bug was logged on 30th June 2014, last updated on 26th June 2017, *and is not yet fixed!*

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
p	The number of physical reads (blocks) that were necessary in order to carry out this FETCH.
cr	The number of consistent reads (blocks) that were necessary in order to carry out this FETCH.
cu	The number of current reads (blocks) that were necessary in order to carry out this FETCH.
mis	Whether this statement was found in the cache (0) or not (1).
r	Rows processed.
dep	Recursion level. 0 = Top-level, user, SQL.
og	Optimiser goal. 1 = ALL_ROWS, 2 = FIRST_ROWS, 3 = RULE, 4 = CHOOSE. Depending on your version of Oracle, you may not see some of the above.
plh	Execution plan hash value.
tim	Time, in microseconds, that this one fetch was completed.

Table 4.5: fetch - Fields

The fields for a FETCH are also the same as those for a PARSE and EXEC.

The mis field is *interesting*. Surely only a PARSE would indicate a miss in the cache? Again, I think Oracle are using the same format of a trace line for PARSE, EXEC and FETCH and simply ignoring or defaulting certain, non-applicable, fields.

You may wish to refer back to page 18 for details of the p, cr and cu block statistics in the EXEC line.

4.6 WAIT

This is the meat of the trace file, usually. Under normal circumstances, it's usual for the various phases of execution (`PARSE`, `EXEC` and `FETCH`) to encounter some `WAIT` events. And I would say, personally[2], that these `WAIT` states are where the vast majority of the total response time is encountered.

A `WAIT` line in the trace file will resemble the following example, up until Oracle 9i at the latest:

```
WAIT #3220341128: nam='db file sequential read' ela= 2769 p1=37 p2
    ⟹ =12931 p3=1
```

Listing 4.8: Wait Line - Oracle 9i

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
nam	The name of the wait event encountered.
ela	Elapsed wall clock time, in microseconds.
p1	Event parameter 1.
p2	Event parameter 2.
p3	Event parameter 3.

Table 4.6: Wait - Fields for Oracle 9i

The `p1`, `p2` and `p3` parameters are different depending on the wait event name. You may require to look them up in `V$EVENT_NAME` to understand what they refer to. For the event above, we see the following:

- p1 is the file number;
- p2 is the (starting) block number;
- p3 is the number of blocks requested.

You may wish to use the following query to determine the various parameters for the `WAIT` events seen in the trace file:

```sql
select name, parameter1, parameter2, parameter3
from   v$event_name
where  name = 'db file sequential read';
```

Listing 4.9: Extracting Event Names for Oracle 9i

Substitute the appropriate event name for your particular `WAIT` event of course. And, don't forget, it *is* case sensitive - Oracle are seriously inconsistent when it comes to naming events!

In Oracle versions 10g onwards, the format is much easier to understand as the generic `p1`, `p2` and `p3` have been replaced by something more meaningful, as follows for the same `WAIT` event:

```
WAIT #3220341128: nam='db file sequential read' ela= 1023 file#=3 block
    ⟹ #=12 blocks=1 obj#=-1 tim=3520817183625
```

Listing 4.10: Wait Line - Oracle 10g Onwards

[2]Beware of small sample sizes!

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
nam	The name of the wait event encountered.
ela	Elapsed wall clock time, in microseconds.
file#	For this wait, the file number.
block#	For this wait, the (starting) block number.
blocks	For this wait, the number of blocks requested.
obj#	If appropriate, the object in question.
tim	Time, in microseconds, that this one wait was completed.

Table 4.7: Parse - Fields for Oracle 10g Onwards

Note The above is an example of one particular `WAIT` event. Other events will have different parameters, as appropriate for the particular event. At least, from 10g onwards, we don't have to look up `V$WAIT_NAME` every time we find a new event in our trace file. Oracle 11g added its own fields too.

The `obj#` refers to a specific object, where one is involved. In the above example, the value is -1, so there is no particular object in this `WAIT`. We are waiting for a block off of the disc, which *may* be part of an index or a table, etc, but for the sake of this `WAIT`, Oracle considers that there is not a specific object involved.

Where an `obj#` is not -1, then the value refers to the `OBJECT_ID` in the `DBA_OBJECTS` view.

4.7 ERROR

If an error is detected while tracing a session, the trace file may show something like the following:

```
ERROR #275452960:err=31013 tim=1075688943194
```

Listing 4.11: Error Line

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
err	The Oracle error code that caused the error.
tim	Time, in microseconds, when this error was detected.

Table 4.8: Error - Fields

In this example, the error relates to `ORA-31013: Invalid XPATH expression` and indeed, is what the user saw on their terminal at the time - and was the reason I traced it because the application wasn't good enough in explaining which object caused the problem!

`ERROR` lines relate to an `EXEC` or `FETCH` operations. `PARSE`s have their own `PARSE ERROR` lines in the trace file.

4.8 STAT

In order to see the `STAT` lines in your trace file, the `TIMED_STATISTICS` parameter for the database must be set to `TRUE`.

In Oracle 9i, there are no execution statistics or timings displayed in the op field. These are only present from 10g onwards.

The `STAT` lines show you *exactly* how Oracle went about getting the data back to you. This is because, what you are seeing is the actual `EXECUTION PLAN` and it is possible for this to be different for the plan displayed by the `EXPLAIN PLAN FOR ...` statement.

Note

In a trace file, the execution plan shows what did happen, as opposed to the explain plan showing what was planned to happen, but may have changed as *stuff* was encountered during the exection.

A typical `STAT` output could resemble the following:

```
STAT #5141189408 id=1 cnt=1 pid=0 pos=1 obj=20 op='TABLE ACCESS BY
    ⟹ INDEX ROWID ICOL$ (cr=4 pr=0 pw=0 time=26 us cost=2 size=54 card
    ⟹ =2)'
STAT #5141189408 id=2 cnt=1 pid=1 pos=1 obj=42 op='INDEX RANGE SCAN
    ⟹ I_ICOL1 (cr=3 pr=0 pw=0 time=22 us cost=1 size=0 card=2)'
```

Listing 4.12: Stat Line

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
id	The identifier for this row of the explain plan may be referred to by the `pid` field, in nested `STAT` lines. Allows a hierarchy to be built.
cnt	The number of rows processed by this step in explain plan.
pid	The parent ID for this step on the plan. Should be zero on the `id=1` line.
pos	The position of this step, within the parent steps in the plan.
obj	The object identifier.
op	The operation performed by this step. This will include additional statistical and timing figures from 10g onwards. Oracle 9i is sadly deficient in this matter!

Table 4.9: Stat - Fields for Oracle 9i

Although the example `STAT` lines above show only a couple of lines, it is possible for more than one to be present under a single parent step. In this case, the `pid` would be the same, but the `pos` would be different.

Warn

`Pos` *should* be a sequentiality increasing number, showing which step was executed when in order to facilitate the parent step, however, I have seen trace files where there have been more than one row with the same `pid` *and* the same `pos`. This is *a bad thing* when trying to figure out what happened when!

In 10g and above, the additional fields in the `op` field itself, are as follows:

Code	Description
cr	The number of consistent reads.
pr	The number of physical reads.
pw	The number of physical writes.
time	The elapsed time in microseconds.
cost	The cost of this step, as determined by the Cost Based Optimiser.
size	From 11g onwards, an estimate of the size, in bytes, of the data returned by this step.
card	The number of rows processed. 11g onwards.

Table 4.10: Stat - Fields for Oracle 10g Onwards

4.9 CLOSE

An example of a CLOSE line from a trace file is as follows:

```
CLOSE #3220452784:c=0,e=13,dep=0,type=0,tim=3520822918452
```

Listing 4.13: Close Line

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
dep	Recursion level. 0 = Top-level, user, SQL.
type	Close type.
tim	Time, in microseconds, that this statement was closed. This is the time it was actually closed **not** when the close started.

Table 4.11: Close - Fields

This line is written when a cursor used for an SQL statement, is no longer required and has been closed. The elapsed times relate to the time it took to close the cursor.

The type field is used to determine how the cursor was closed. It takes the following values:

- 0 if the cursor was hard closed. This indicates that the cursor was not saved in the server side closed cursor cache for later reuse. This can be because:
  - The statement is a DDL statement. DDL statements are never cached.
  - SESSION_CACHED_CURSORS is set to zero so no caching is permitted.
  - The statement *could* be cached, but as it has not been executed often enough (three times minimum), then it has not yet been cached.
- 1 if the cursor has been cached, as opposed to properly closed, in an empty slot in the cache.
- 2 if the cursor was cached, but caused another cursor to be aged out as there were no free slots.
- 3 if the cursor was used from the cache, and on CLOSE, remains cached.

A cursor that was cached on CLOSEing may be reused for the same SQL statement at a later time during the session, in this case, there will not be a PARSE for the statement prior to the next BINDS or EXEC for the statement as it was not closed after the previous usage. When Oracle came to PARSE the SQL again, it was found to be in the cache, and thus, was still open from the previous usage.

A cursor ID that has been hard CLOSEd may be re-used by a subsequent opening of a new cursor, which can be for a different statement, or for this one again.

4.10 XCTEND

This line in a trace file indicates the end of a transaction. Sometimes you will see this:

```
PARSING IN CURSOR #398131288 len=6 dep=1 uid=90 oct=44 lid=90 tim
    ⟹ =1484913807072 hv=255718823 ad='0' sqlid='8ggw94h7mvxd7'
COMMIT
END OF STMT
PARSE #398131288:c=0,e=4,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=0,plh=0,tim
    ⟹ =1484913807071
XCTEND rlbk=0, rd_only=1, tim=1484913807100
EXEC #398131288:c=0,e=27,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=0,plh=0,tim
    ⟹ =1484913807121
CLOSE #398131288:c=0,e=0,dep=1,type=3,tim=1484913807132
```

Listing 4.14: Commit Statement with Xctend Line

You can see here the entire `PARSE`, `EXEC` and `CLOSE` for the statement. Sometimes, you don't see the `COMMIT` (or `ROLLBACK`) statement being parsed, all you see is the `XCTEND` line in the trace file. Why? I have no idea - it's an Oracle thing!

You should notice in the above, that the transaction was completed *before* the `EXEC` completed. remember that the trace file shows things after they have finished. So the `EXEC` of the `COMMIT` statement had to wait for the transaction to end before it could complete.

The fields you will see in an `XCTEND` line are as follows:

Code	Description
rlbk	0 indicates `COMMIT`, 1 is for `ROLLBACK`.
rd_only	Whether or not the transaction was read only (1) or read write (0) regardless of `COMMIT` or `ROLLBACK` being executed to end it.
tim	Time, in microseconds, at which the transaction ended.

Table 4.12: Xctend - Fields

4.11 BINDS

To see the bind details in a trace file, you need to have enabled at least trace level 4 (for a 10046 event), or set `binds => true` for `DBMS_SUPPORT` and `DBMS_MONITOR` calls to start tracing. You should also be *using* binds in your SQL statement too of course!

There is a *lot* of information in the binds section of a trace file. For 9i, the following data are listed:

Code	Description
dty	Data type code. (See Appendices.)
mxl	Maximum length of the bind variable value. (Private maximum length in parentheses.)
mal	Array length. Only for array binds.
scl	Scale. Only for numeric binds.
pre	Precision. Only for numeric binds.
oacflg	Special flag indicating bind options.
oacflg2	Second part of oacflg.
size	Amount of memory to be allocated for this chunk of the bind.
offset	Offset into this chunk for this bind buffer.
bfp	Bind address.
bln	Bind buffer length.
avl	Actual value or array length. A value of zero = NULL, or a PL/SQL OUT bind - if the cursor's command is 47 for PL/SQL Execution.
flg	Bind status flag.
value	Value of the bind variable.

Table 4.13: Binds - Fields for Oracle 9i

While for 10g upwards, we would expect to see the following:

Code	Description
oacdty	Data type code. (See Appendices.)
mxl	Maximum length, in bytes, of the bind variable value. (Used length in parentheses.)
mxlc	Maximum length, in characters, of the bind variable value. (Used length in parentheses.)
mal	Array length. Only for array binds.
scl	Scale. Only for numeric binds.
pre	Precision. Only for numeric binds.
oacflg	Special flag indicating bind options.
fl2	Second part of oacflg.
frm	Unknown.
csi	Identifier code for the database's default or national character set. (See Appendices.) Only used in character/string binds.
siz	Size of memory to be allocated for this chunk.
off	Offset into the chunk of the bind buffer.
kxsbbbfp	Bind address.

Table 4.14: Binds - Fields for Oracle 10g Onwards… *continues on next page*

Code	Description
bln	Bind buffer length.
avl	Actual value or array length. A value of zero = NULL, or a PL/SQL OUT bind - if the cursor's command is 47 for PL/SQL Execution.
flg	Bind status flag.
value	Value of the bind variable, or a memory dump. This field will only be shown for any bind that has a non-NULL value.

Table 4.14: Binds - Fields for Oracle 10g Onwards

Binds always number from left to right in an SQL statement. This means that the first bind found in the statement, will be listed in the trace file as `BIND#0` regardless of its actual name or number in the statement.

If a bind, for example `:3`, is used more than once in the *same* SQL statement, then it will appear *once* in the binds list - as `BIND#2`, however, the statement itself will refer to it correctly as `:3` each time it is used.

The `oacdty` values are listed in the appendices for reference. There are quite a few of these, and you should note that experience has shown that the Oracle documentation doesn't always seem match up to the reality of a trace file.

Always be aware, however, that just because a bind is defined to be a VARCHAR2, `oacdty=01`, for example, it doesn't mean that the column in the table it relates to is also a VARCHAR2. Some people write code that passes VARCHAR2 values to a DATE column - this negating the ability to ever use an index (unless a function based index is created) on that DATE column!

The `mxl` fields shows how big the buffer assigned to this bind variable is, and how much of it has been used. This is measured in bytes. The `mxlc` shows, where applicable, the buffer size and used size in characters[3]. This will depend on the character set in use. If the character size is not appropriate, then `mxlc` will be zero.

4.11.1 Examples

The following is an 11g example of the binds section for the following (recursive) SQL statement:

```sql
select  o.owner#, o.name, o.namespace, o.obj#, d.d_timestamp,
        nvl(d.property,0), o.type#, o.subname, d.d_attrs
from    dependency$ d, obj$ o
where   d.p_obj#=:1
and     (d.p_timestamp=nvl(:2,d.p_timestamp) or d.property=2)
and     o.owner#=nvl(:3,o.owner#)
and     d.d_obj#=o.obj#
order   by o.obj#
```

Listing 4.15: Example of Recursive SQL Statement

The binds section, extracted from the trace file, looks as follows:

```
BINDS #741210192:
 Bind#0
```

[3] At least, that's what *I* think it shows!

```
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=00 fl2=0001 frm=00 csi=00 siz=56 off=0
  kxsbbbfp=2c2d4c58  bln=22  avl=04  flg=05
  value=104305
 Bind#1
  oacdty=12 mxl=07(07) mxlc=00 mal=00 scl=00 pre=00
  oacflg=11 fl2=0001 frm=00 csi=00 siz=0 off=24
  kxsbbbfp=2c2d4c70  bln=07  avl=07  flg=01
  value="6/26/2017 9:58:26"
 Bind#2
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=01 fl2=0001 frm=00 csi=00 siz=0 off=32
  kxsbbbfp=2c2d4c78  bln=22  avl=00  flg=01
```

Listing 4.16: Binds Lines

Looking specifically at the above, we can see the following:

- We have two `NUMBER` binds, and one `DATE`;
- Neither of the `NUMBER`s have a scale or precision;
- The last bind, `BIND#2`, is a `NUMBER` and is `NULL` (`avl=00`) and so has no value clause;
- Because these binds are of specific types, where the storage is always exactly as indicated by the `mxl` field, the private storage used for the value is always the same length. In other words, `NUMBER` data types are always 22 bytes while `DATE`s are 7;
- There are no character sets applicable to `NUMBER` or `DATE` data types (`csi=00`).

Character (VARCHAR2, CHAR etc) binds take the following form:

```
Bind#1
 oacdty=01 mxl=32(04) mxlc=00 mal=00 scl=00 pre=00
 oacflg=10 fl2=0001 frm=01 csi=31 siz=0 off=24
 kxsbbbfp=610cd550 bln=32 avl=04 flg=01
 value="DUAL"
```

Listing 4.17: Bind Example - VARCHAR2 with WE8ISO8859P1 Characterset

Here we can see that:

- `csi=31` is listed so checking with the character set list, in the appendices of this document, we see that this means that the bind is using the `WE8ISO8859P1` character set;
- The `value` can be easily read from the trace file. Had this been in a different character set, `ALUTF16` for example (`csi=2000`), then the value would most likely have been dumped in hexadecimal, as follows:

```
...
value = 0 44 0 55 0 41 0 4C
```

Listing 4.18: Bind Example - VARCHAR2 with ALUTF16 Characterset

- We can also see that although the maximum size of the buffer for this bind is 32, (`mxl=32`), only 4 bytes are in use (`mxl=32(04)`).

A `REF_CURSOR` bind, will resemble the following. It has an `oacdty` of 102, which appears to confuse the `value` field a little, as it implies that it cannot handle a data type of 102. Hmmm.

```
Bind#3
 oacdty=102 mxl=04(04) mxlc=00 mal=00 scl=00 pre=00
 oacflg=01 fl2=1000000 frm=00 csi=00 siz=0 off=176
 kxsbbbfp=c2a91548 bln=04 avl=04 flg=01
```

```
 value=Unhandled datatype (102) found in kxsbndinf
Dump of memory from 0x00000000C2A91548 to 0x00000000C2A9154C
0C2A91540 00000000 [....]
```

Listing 4.19: Bind Example - REF_CURSOR

4.12 UNMAP

If the cursor in question used a temporary table then you will see an UNMAP in the trace file when the cursor is CLOSEd and the locks on the temporary table are freed. The UNMAP covers the following stages of operation:

- Free the lock;
- Delete the state object;
- Free the temporary segment.

If you use to analyse trace files, the UNMAP data are accumulated into the EXECUTE statistics for the cursor. (See Oracle note 39817.1.)

Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
p	The number of physical reads (blocks) that were necessary in order to carry out this PARSE.
cr	The number of consistent reads (blocks) that were necessary in order to carry out this PARSE.
cu	The number of current reads (blocks) that were necessary in order to carry out this PARSE.
mis	Whether this statement was found in the cache (0) or not (1). Indicates whether or not a hard parse was required. Seeing a 1 is bad, usually.
r	Rows processed.
dep	Recursion level. 0 = Top-level, user, SQL.
og	Optimiser goal. 1 = ALL_ROWS, 2 = FIRST_ROWS, 3 = RULE, 4 = CHOOSE. Depending on your version of Oracle, you may not see some of the above.
plh	Execution plan hash value.
tim	Time, in microseconds, when the parse for the statement completed. **Not** the time it took.

Table 4.15: Parse - Fields

The plh value is a value that corresponds to the column PLAN_HASH_VALUE in VSQL_PLAN, VSQL_PLAN_STATISTICS_ALL and V$SQLSTATS. (There may be other views where this value appears, depending on the Oracle version in use.)

UNMAP trace lines, if you ever see one, are identical to those of PARSE, EXEC, FETCH as described above, and SORT UNMAP described below.

These lines in the trace file are related to cleaning up the temporary segments in any temporary tables used by the cursor.

4.13 SORT UNMAP

`SORT UNMAP` lines in the trace file are similar to the `UNMAP` lines described above. These, however, relate to the freeing up of temporary segments used for a sort operations.

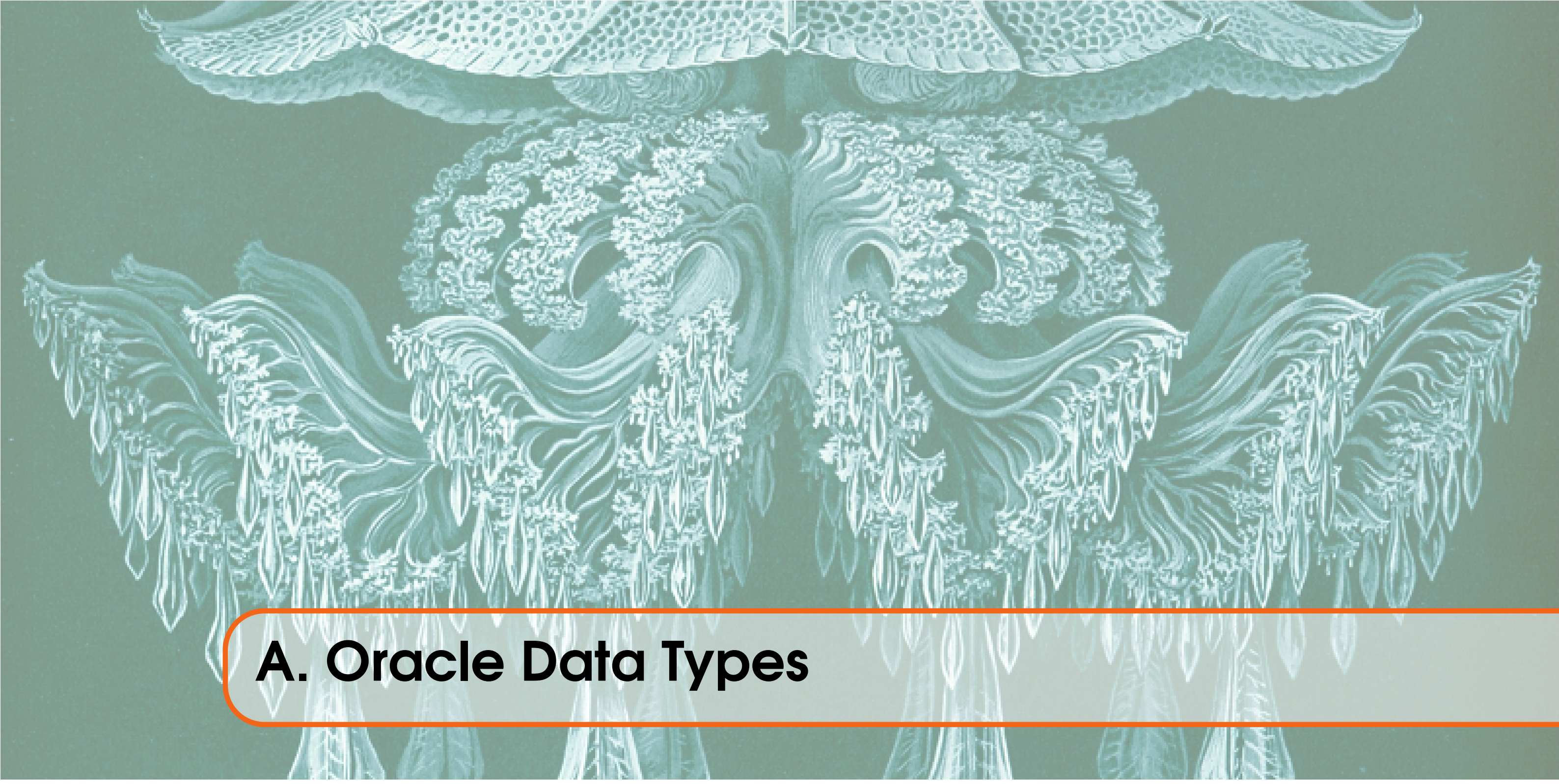
Code	Description
#nnnn	The cursor ID. This may be reused if for future cursors if this one is closed, and another opened.
c	Elapsed CPU time. Microseconds.
e	Elapsed wall clock time, also in microseconds.
p	The number of physical reads (blocks) that were necessary in order to carry out this `PARSE`.
cr	The number of consistent reads (blocks) that were necessary in order to carry out this `PARSE`.
cu	The number of current reads (blocks) that were necessary in order to carry out this `PARSE`.
mis	Whether this statement was found in the cache (0) or not (1). Indicates whether or not a hard parse was required. Seeing a 1 is bad, usually.
r	Rows processed.
dep	Recursion level. 0 = Top-level, user, SQL.
og	Optimiser goal. 1 = ALL_ROWS, 2 = FIRST_ROWS, 3 = RULE, 4 = CHOOSE. Depending on your version of Oracle, you may not see some of the above.
plh	Execution plan hash value.
tim	Time, in microseconds, when the parse for the statement completed. **Not** the time it took.

Table 4.16: Parse - Fields

The `plh` value is a value that corresponds to the column `PLAN_HASH_VALUE` in `V$SQL_PLAN`, `V$SQL_PLAN_STATISTICS_ALL` and `V$SQLSTATS`. (There may be other views where this value appears, depending on the Oracle version in use.)

`SORT UNMAP` trace lines, if you ever see one, are identical to those of `PARSE`, `EXEC`, `FETCH` and `UNMAP` as described above. They are related to cleaning up the sort segments whose details can be seen in `V$SORT_USAGE`. This applies to both temporary segments in memory or actual disc based segments where a large sort has overflowed memory onto disc.

II Appendices



A. Oracle Data Types

Bind Data Types

The `oacdty` parameter in a bind variables details determines the data type of that bind variable. This is not necessarily the data type of the column in a table that it may be being `INSERT`ed or `UPDATE`d into, or compared against in a `WHERE` clause.

There is a table of data types and codes in the *Internal Data Types* section of the *Data Types* chapter in the 12cR2 *Oracle Call Interface* manual.

However, various other sources on the internet, and in books, seem to disagree with some of what the above table shows. In addition, I have come across many Oracle Trace files where a ROWID was code 11 and not code 69. Consistency? Who mentioned consistency?

In 11g (possibly 10g) onwards, there is a new view named `V$SQL_BIND_CAPTURE` which lists various details about binds variables for SQL cursors. Two columns are of interest here, `DATATYPE` and `DATATYPE_STRING` which we can use to convert a data type number into an actual data type name. In addition, we can look at the source code for the view named `DBA_TAB_COLS`, or renamed to `DBA_TAB_COLS_V$` in 12c.

Code	Data Type
0	UNDEFINED
1	NVARCHAR2 or VARCHAR2
2	NUMBER or FLOAT(precision)
8	LONG
9	NCHAR VARYING or VARCHAR
11	ROWID
12	DATE
16	DATE
23	RAW

Table A.1: Bind Variable data Types… *continues on next page*

Code	Data Type
24	LONG RAW
25	UB2
29	B4
58	ANYDATA, XMLTYPE or OPAQUE
69	ROWID
96	NCHAR or CHAR
100	BINARY_FLOAT
101	BINARY_DOUBLE
102	REF CURSOR
105	MLSLABEL
106	MLSLABEL
108	User defined Types
111	REF XMLTYPE
112	NCLOB or CLOB
113	BLOB
114	BFILE
115	CFILE
121	User defined and/or object TYPEs
122	User defined and/or object TYPEs, NESTED TABLE
123	User defined and/or object TYPEs, VARRAY
178	TIME(scale)
179	TIME(scale) WITH TIME ZONE
180	TIMESTAMP(scale)
181	TIMESTAMP(scale) WITH TIME ZONE
182	INTERVAL YEAR(precision) TO MONTH
183	INTERVAL DAY(precision) TO SECOND(scale)
208	UROWID (Universal ROWID)
231	TIMESTAMP(scale) WITH LOCAL TIME ZONE

Table A.1: Bind Variable data Types

Data type codes in the above table have been extracted from the *Oracle Call Interface* manual, numerous trace files and/or the views mentioned. There are, as noted, inconsistencies between the various sources which is why there are the occasional duplicate entry.Sadly, Oracle doesn't document some internal details as much as it should (or as much as *I* think it should!).

Note Data types 178 and 179 are both listed in 11g and 12c, but you try creating a table with the `TIME(n)` or `TIME(n) WITH TIME ZONE` and see what happens! They *appear* to be Java data types as that appears to be the only manual that lists them as valid data types.

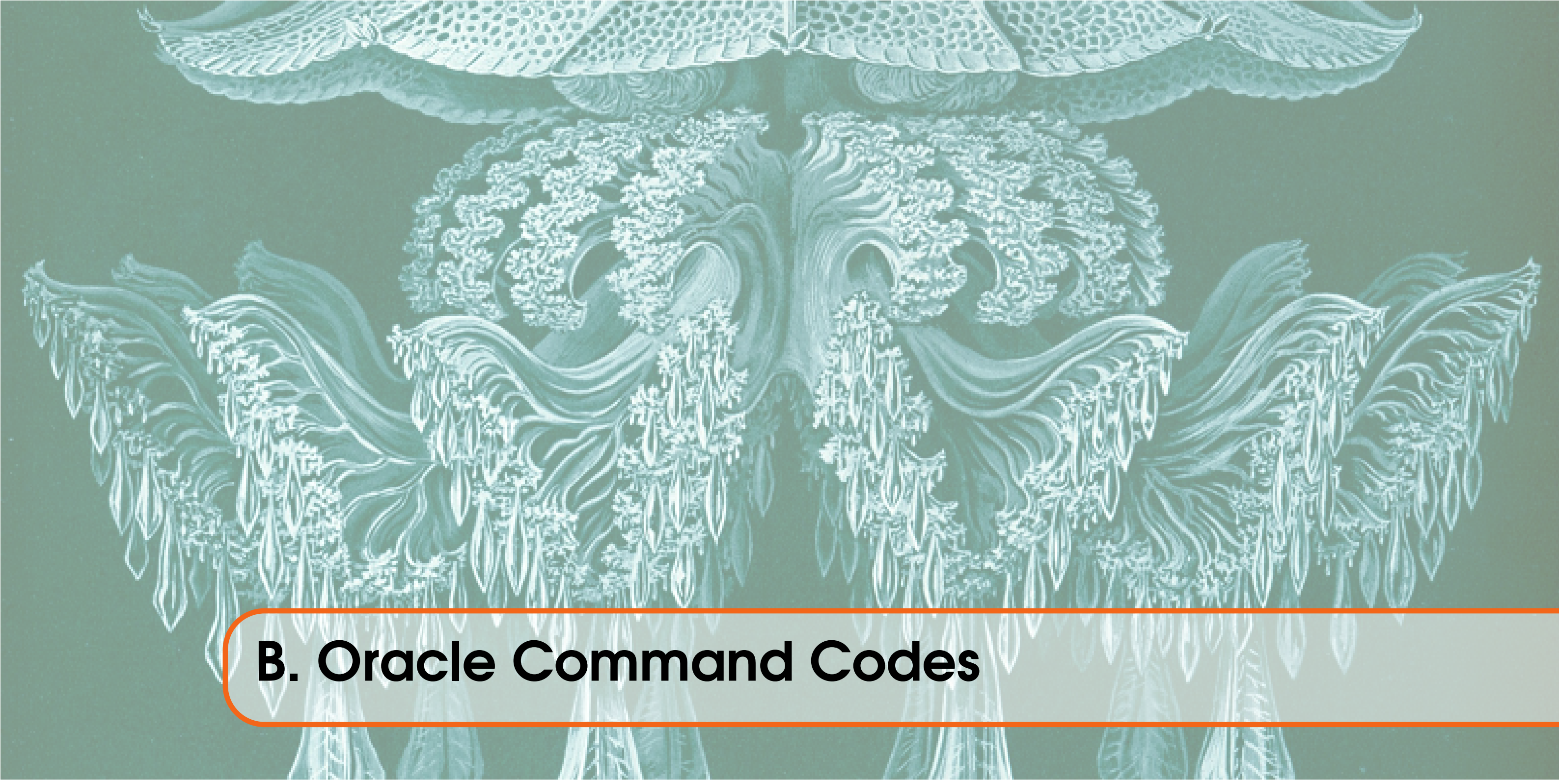
For an example of inconsistencies, here is a `ROWID` with `oacdty=11` as opposed to `oacdty=69`. The following example was taken from a trace file, created on Windows with Oracle 11.2.0.4:

```
PARSING IN CURSOR #5141169152 len=37 dep=1 uid=0 oct=3 lid=0 tim
    ⟹ =3520788504344 hv=1398610540 ad='7ffc0a95d898' sqlid='
    ⟹ grwydz59pu6mc'
select text from view$ where rowid=:1
END OF STMT
```

```
PARSE #5141169152:c=0,e=14640,p=0,cr=29,cu=0,mis=1,r=0,dep=1,og=4,plh
    ⟹ =0,tim=3520788504343
BINDS #5141169152:
 Bind#0
  oacdty=11 mxl=16(16) mxlc=00 mal=00 scl=00 pre=00
  oacflg=18 fl2=0001 frm=00 csi=00 siz=16 off=0
  kxsbbbfp=bff30890  bln=16  avl=16  flg=05
  value=00002787.000A.0001
```

Listing A.1: Bind Example - `ROWID` With `oacdty=11`

There's also an example of a `REF_CURSOR` bind on page 32 of this eBook showing the use of an `oacdty=102`.



B. Oracle Command Codes

Command Codes

The `oct` parameter in a `PARSING IN CURSOR` line in an Oracle trace file, determines the command that is being parsed in the SQL statement. Why we should need this, I have no idea, as the SQL text for the command will - obviously - show what command is being parsed. However…

The following (large) table outlines the various command codes and was extracted from an Oracle 12.1.0.2 database.

Code	Command	Code	Command
0	UNKNOWN	111	DROP PUBLIC SYNONYM
1	CREATE TABLE	112	CREATE PUBLIC DATABASE LINK
2	INSERT	113	DROP PUBLIC DATABASE LINK
3	SELECT	114	GRANT ROLE
4	CREATE CLUSTER	115	REVOKE ROLE
5	ALTER CLUSTER	116	EXECUTE PROCEDURE
6	UPDATE	117	USER COMMENT
7	DELETE	118	ENABLE TRIGGER
8	DROP CLUSTER	119	DISABLE TRIGGER
9	CREATE INDEX	120	ENABLE ALL TRIGGERS
10	DROP INDEX	121	DISABLE ALL TRIGGERS
11	ALTER INDEX	122	NETWORK ERROR
12	DROP TABLE	123	EXECUTE TYPE
13	CREATE SEQUENCE	128	FLASHBACK
14	ALTER SEQUENCE	129	CREATE SESSION
15	ALTER TABLE	130	ALTER MINING MODEL
16	DROP SEQUENCE	131	SELECT MINING MODEL
17	GRANT OBJECT	133	CREATE MINING MODEL

Table B.1: Oracle Command Codes… *continues on next page*

Code	Command	Code	Command
18	REVOKE OBJECT	134	ALTER PUBLIC SYNONYM
19	CREATE SYNONYM	135	DIRECTORY EXECUTE
20	DROP SYNONYM	136	SQL*LOADER DIRECT PATH LOAD
21	CREATE VIEW	137	DATAPUMP DIRECT PATH UNLOAD
22	DROP VIEW	138	DATABASE STARTUP
23	VALIDATE INDEX	139	DATABASE SHUTDOWN
24	CREATE PROCEDURE	140	CREATE SQL TXLN PROFILE
25	ALTER PROCEDURE	141	ALTER SQL TXLN PROFILE
26	LOCK	142	USE SQL TXLN PROFILE
27	NO-OP	143	DROP SQL TXLN PROFILE
28	RENAME	144	CREATE MEASURE FOLDER
29	COMMENT	145	ALTER MEASURE FOLDER
30	AUDIT OBJECT	146	DROP MEASURE FOLDER
31	NOAUDIT OBJECT	147	CREATE CUBE BUILD PROCESS
32	CREATE DATABASE LINK	148	ALTER CUBE BUILD PROCESS
33	DROP DATABASE LINK	149	DROP CUBE BUILD PROCESS
34	CREATE DATABASE	150	CREATE CUBE
35	ALTER DATABASE	151	ALTER CUBE
36	CREATE ROLLBACK SEG	152	DROP CUBE
37	ALTER ROLLBACK SEG	153	CREATE CUBE DIMENSION
38	DROP ROLLBACK SEG	154	ALTER CUBE DIMENSION
39	CREATE TABLESPACE	155	DROP CUBE DIMENSION
40	ALTER TABLESPACE	157	CREATE DIRECTORY
41	DROP TABLESPACE	158	DROP DIRECTORY
42	ALTER SESSION	159	CREATE LIBRARY
43	ALTER USER	160	CREATE JAVA
44	COMMIT	161	ALTER JAVA
45	ROLLBACK	162	DROP JAVA
46	SAVEPOINT	163	CREATE OPERATOR
47	PL/SQL EXECUTE	164	CREATE INDEXTYPE
48	SET TRANSACTION	165	DROP INDEXTYPE
49	ALTER SYSTEM	166	ALTER INDEXTYPE
50	EXPLAIN	167	DROP OPERATOR
51	CREATE USER	168	ASSOCIATE STATISTICS
52	CREATE ROLE	169	DISASSOCIATE STATISTICS
53	DROP USER	170	CALL METHOD
54	DROP ROLE	171	CREATE SUMMARY
55	SET ROLE	172	ALTER SUMMARY
56	CREATE SCHEMA	173	DROP SUMMARY
57	CREATE CONTROL FILE	174	CREATE DIMENSION
59	CREATE TRIGGER	175	ALTER DIMENSION
60	ALTER TRIGGER	176	DROP DIMENSION
61	DROP TRIGGER	177	CREATE CONTEXT
62	ANALYZE TABLE	178	DROP CONTEXT
63	ANALYZE INDEX	179	ALTER OUTLINE
64	ANALYZE CLUSTER	180	CREATE OUTLINE
65	CREATE PROFILE	181	DROP OUTLINE

Table B.1: Oracle Command Codes… *continues on next page*

Code	Command	Code	Command
66	DROP PROFILE	182	UPDATE INDEXES
67	ALTER PROFILE	183	ALTER OPERATOR
68	DROP PROCEDURE	190	PASSWORD CHANGE
70	ALTER RESOURCE COST	192	ALTER SYNONYM
71	CREATE MATERIALIZED VIEW LOG	197	PURGE USER_RECYCLEBIN
72	ALTER MATERIALIZED VIEW LOG	198	PURGE DBA_RECYCLEBIN
73	DROP MATERIALIZED VIEW LOG	199	PURGE TABLESPACE
74	CREATE MATERIALIZED VIEW	200	PURGE TABLE
75	ALTER MATERIALIZED VIEW	201	PURGE INDEX
76	DROP MATERIALIZED VIEW	202	UNDROP OBJECT
77	CREATE TYPE	204	FLASHBACK DATABASE
78	DROP TYPE	205	FLASHBACK TABLE
79	ALTER ROLE	206	CREATE RESTORE POINT
80	ALTER TYPE	207	DROP RESTORE POINT
81	CREATE TYPE BODY	208	PROXY AUTHENTICATION ONLY
82	ALTER TYPE BODY	209	DECLARE REWRITE EQUIVALENCE
83	DROP TYPE BODY	210	ALTER REWRITE EQUIVALENCE
84	DROP LIBRARY	211	DROP REWRITE EQUIVALENCE
85	TRUNCATE TABLE	212	CREATE EDITION
86	TRUNCATE CLUSTER	213	ALTER EDITION
88	ALTER VIEW	214	DROP EDITION
91	CREATE FUNCTION	215	DROP ASSEMBLY
92	ALTER FUNCTION	216	CREATE ASSEMBLY
93	DROP FUNCTION	217	ALTER ASSEMBLY
94	CREATE PACKAGE	218	CREATE FLASHBACK ARCHIVE
95	ALTER PACKAGE	219	ALTER FLASHBACK ARCHIVE
96	DROP PACKAGE	220	DROP FLASHBACK ARCHIVE
97	CREATE PACKAGE BODY	225	ALTER DATABASE LINK
98	ALTER PACKAGE BODY	226	CREATE PLUGGABLE DATABASE
99	DROP PACKAGE BODY	227	ALTER PLUGGABLE DATABASE
100	LOGON	228	DROP PLUGGABLE DATABASE
101	LOGOFF	229	CREATE AUDIT POLICY
102	LOGOFF BY CLEANUP	230	ALTER AUDIT POLICY
103	SESSION REC	231	DROP AUDIT POLICY
104	SYSTEM AUDIT	232	CODE-BASED GRANT
105	SYSTEM NOAUDIT	233	CODE-BASED REVOKE
106	AUDIT DEFAULT	238	ADMINISTER KEY MANAGEMENT
107	NOAUDIT DEFAULT	239	CREATE MATERIALIZED ZONEMAP
108	SYSTEM GRANT	240	ALTER MATERIALIZED ZONEMAP
109	SYSTEM REVOKE	241	DROP MATERIALIZED ZONEMAP
110	CREATE PUBLIC SYNONYM	305	ALTER PUBLIC DATABASE LINK

Table B.1: Oracle Command Codes

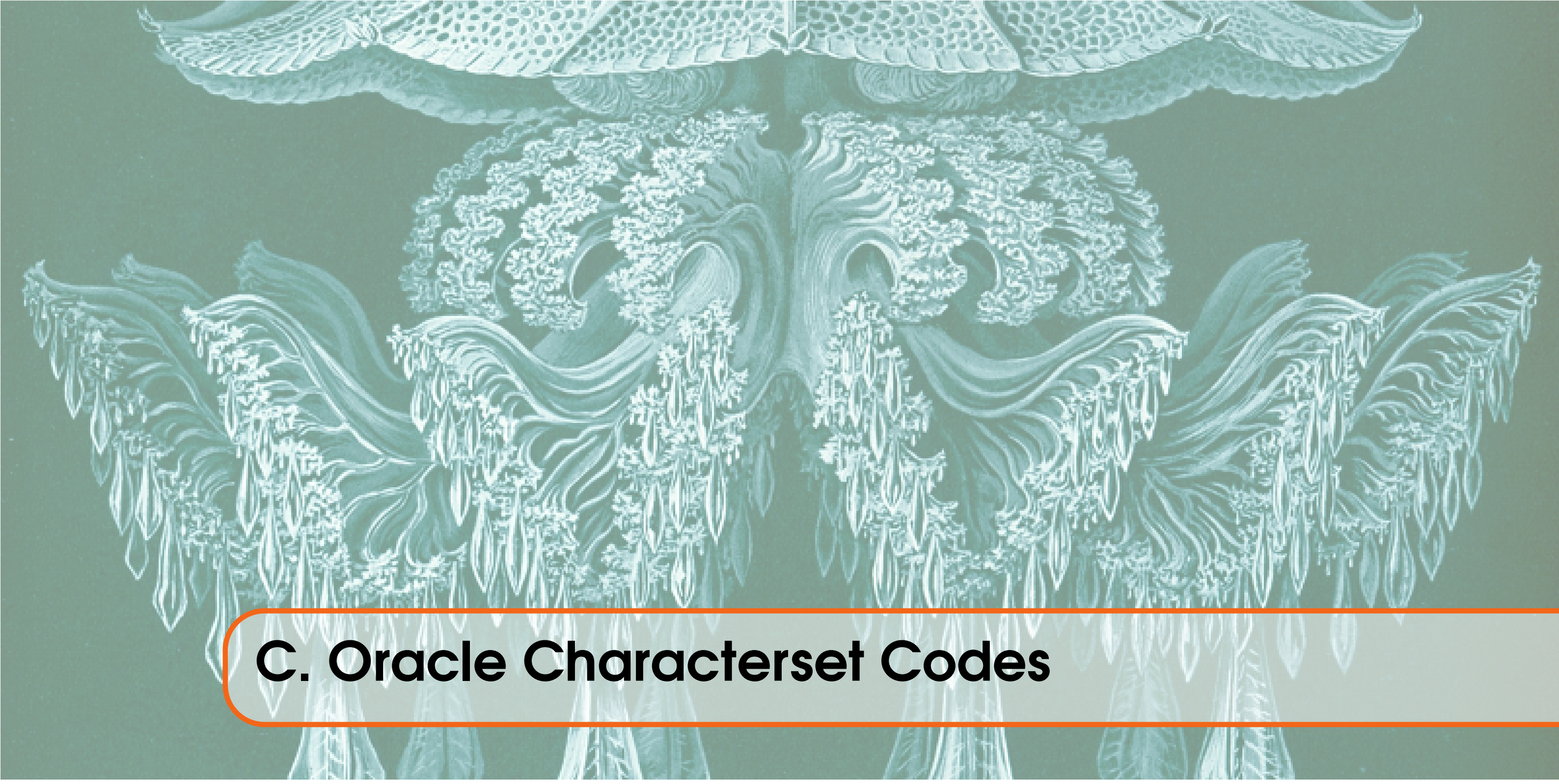
The exact list of commands for your particular database version can be extracted using the following SQL command:

```
select action as code,
```

```
        name as command
from    audit_actions;
```

Listing B.1: SQL Query to List Oracle Command Codes

There are 212 different commands in Oracle 12c (12.1.0.2) while Oracle 11g (11.2.0.4) has only (!) 181.



C. Oracle Characterset Codes

Bind Charactersets

Some data types use different character sets. These are coded in the `csi` field in the bind details lines of the trace file. You can extract the list of current character set codes and names with the following query:

```sql
select  nls_charset_id(value), value
from    v$nls_valid_values
where   isdeprecated='FALSE'
and     parameter = 'CHARACTERSET'
order   by nls_charset_id(value);
```

Listing C.1: SQL Query to list Character Set Codes and Names

The values that you may see here are as follows, taken from an Oracle 12.1.0.1 database where there are 222 character sets listed:

Code	Character Set	Code	Character Set	Code	Character Set
1	US7ASCII	159	CL8MACCYRILLICS	301	EE8EBCDIC870C
2	WE8DEC	160	WE8PC860	312	TR8EBCDIC1026S
3	WE8HP	161	IS8PC861	314	BLT8EBCDIC1112S
4	US8PC437	162	EE8MACCES	315	IW8EBCDIC424S
5	WE8EBCDIC37	163	EE8MACCROATIANS	316	EE8EBCDIC870S
6	WE8EBCDIC500	164	TR8MACTURKISHS	317	CL8EBCDIC1025S
7	WE8EBCDIC1140	165	IS8MACICELANDICS	319	TH8TISEBCDICS
8	WE8EBCDIC285	166	EL8MACGREEKS	320	AR8EBCDIC420S
9	WE8EBCDIC1146	167	IW8MACHEBREWS	322	CL8EBCDIC1025C
10	WE8PC850	170	EE8MSWIN1250	323	CL8EBCDIC1025R
11	D7DEC	171	CL8MSWIN1251	324	EL8EBCDIC875R

Table C.1: Oracle Character Set Codes. . . *continues on next page*

Code	Character Set	Code	Character Set	Code	Character Set
12	F7DEC	172	ET8MSWIN923	325	CL8EBCDIC1158
13	S7DEC	173	BG8MSWIN	326	CL8EBCDIC1158R
14	E7DEC	174	EL8MSWIN1253	327	EL8EBCDIC423R
15	SF7ASCII	175	IW8MSWIN1255	351	WE8MACROMAN8
16	NDK7DEC	176	LT8MSWIN921	352	WE8MACROMAN8S
17	I7DEC	177	TR8MSWIN1254	353	TH8MACTHAI
18	NL7DEC	178	WE8MSWIN1252	354	TH8MACTHAIS
19	CH7DEC	179	BLT8MSWIN1257	368	HU8CWI2
20	YUG7ASCII	180	D8EBCDIC273	380	EL8PC437S
21	SF7DEC	181	I8EBCDIC280	381	EL8EBCDIC875
22	TR7DEC	182	DK8EBCDIC277	382	EL8PC737
23	IW7IS960	183	S8EBCDIC278	383	LT8PC772
25	IN8ISCII	184	EE8EBCDIC870	384	LT8PC774
27	WE8EBCDIC1148	185	CL8EBCDIC1025	385	EL8PC869
28	WE8PC858	186	F8EBCDIC297	386	EL8PC851
31	WE8ISO8859P1	187	IW8EBCDIC1086	390	CDN8PC863
32	EE8ISO8859P2	188	CL8EBCDIC1025X	401	HU8ABMOD
33	SE8ISO8859P3	189	D8EBCDIC1141	500	AR8ASMO8X
34	NEE8ISO8859P4	190	N8PC865	554	AR8NAFITHA711
35	CL8ISO8859P5	191	BLT8CP921	555	AR8SAKHR707
36	AR8ISO8859P6	192	LV8PC1117	556	AR8MUSSAD768
37	EL8ISO8859P7	193	LV8PC8LR	557	AR8ADOS710
38	IW8ISO8859P8	194	BLT8EBCDIC1112	558	AR8ADOS720
39	WE8ISO8859P9	195	LV8RST104090	559	AR8APTEC715
40	NE8ISO8859P10	196	CL8KOI8R	560	AR8MSWIN1256
41	TH8TISASCII	197	BLT8PC775	561	AR8NAFITHA721
42	TH8TISEBCDIC	198	DK8EBCDIC1142	563	AR8SAKHR706
43	BN8BSCII	199	S8EBCDIC1143	565	AR8ARABICMAC
44	VN8VN3	200	I8EBCDIC1144	566	AR8ARABICMACS
45	VN8MSWIN1258	201	F7SIEMENS9780X	590	LA8ISO6937
46	WE8ISO8859P15	202	E7SIEMENS9780X	829	JA16VMS
47	BLT8ISO8859P13	203	S7SIEMENS9780X	830	JA16EUC
48	CEL8ISO8859P14	204	DK7SIEMENS9780X	831	JA16EUCYEN
49	CL8ISOIR111	205	N7SIEMENS9780X	832	JA16SJIS
50	WE8NEXTSTEP	206	I7SIEMENS9780X	833	JA16DBCS
51	CL8KOI8U	207	D7SIEMENS9780X	834	JA16SJISYEN
52	AZ8ISO8859P9E	208	F8EBCDIC1147	835	JA16EBCDIC930
70	AR8EBCDICX	210	WE8GCOS7	836	JA16MACSJIS
81	EL8DEC	211	EL8GCOS7	837	JA16EUCTILDE
82	TR8DEC	221	US8BS2000	838	JA16SJISTILDE
90	WE8EBCDIC37C	222	D8BS2000	840	KO16KSC5601
91	WE8EBCDIC500C	223	F8BS2000	842	KO16DBCS
92	IW8EBCDIC424	224	E8BS2000	845	KO16KSCCS
93	TR8EBCDIC1026	225	DK8BS2000	846	KO16MSWIN949
94	WE8EBCDIC871	226	S8BS2000	850	ZHS16CGB231280
95	WE8EBCDIC284	230	WE8BS2000E	851	ZHS16MACCGB23128
96	WE8EBCDIC1047	231	WE8BS2000	852	ZHS16GBK

Table C.1: Oracle Character Set Codes. . . *continues on next page*

Code	Character Set	Code	Character Set	Code	Character Set
97	WE8EBCDIC1140C	232	EE8BS2000	853	ZHS16DBCS
98	WE8EBCDIC1145	233	CE8BS2000	854	ZHS32GB18030
99	WE8EBCDIC1148C	235	CL8BS2000	860	ZHT32EUC
100	WE8EBCDIC1047E	239	WE8BS2000L5	861	ZHT32SOPS
101	WE8EBCDIC924	241	WE8DG	862	ZHT16DBT
110	EEC8EUROASCI	251	WE8NCR4970	863	ZHT32TRIS
113	EEC8EUROPA3	261	WE8ROMAN8	864	ZHT16DBCS
114	LA8PASSPORT	262	EE8MACCE	865	ZHT16BIG5
140	BG8PC437S	263	EE8MACCROATIAN	866	ZHT16CCDC
150	EE8PC852	264	TR8MACTURKISH	867	ZHT16MSWIN950
152	RU8PC866	265	IS8MACICELANDIC	868	ZHT16HKSCS
153	RU8BESTA	266	EL8MACGREEK	871	UTF8
154	IW8PC1507	267	IW8MACHEBREW	872	UTFE
155	RU8PC855	277	US8ICL	873	AL32UTF8
156	TR8PC857	278	WE8ICL	992	ZHT16HKSCS31
158	CL8MACCYRILLIC	279	WE8ISOICLUK	2000	AL16UTF16

Table C.1: Oracle Character Set Codes

If you see a character set code in the `csi` field, and you don't have the above list, you can determine the characterset in use with the following query:

```
SELECT NLS_CHARSET_NAME(1) FROM dual;
```

Listing C.2: SQL Query to Convert a `csi` Code to a Charcter Set Name

Correspondingly, you can go from a charcter set name to its `csi` code with the following query:

```
SELECT NLS_CHARSET_ID('US7ASCII') FROM dual;
```

Listing C.3: SQL Query to Convert a Charcter Set Name to a `csi` Code

D. How this Book Evolved

D.1 Why this Book?

This eBook originally started as a collection of useful notes, scripts, etc which I had come across, written or deduced, over the years of playing with Oracle Trace Files and various Trace File Analysis tools.

Eventually, I collected them all together in one place so that I could have all the useful information, in one easy to find place[1].

There are many places on the internet, people I have met or worked with, books which I have bought (or had bought for me) and read, to whom I will always be grateful as they have taught me many things. Maybe I can help others as I have been helped.

As Isaac Newton is thought to have said, "*I have stood on the shoulders of giants*" - me too.

D.2 Creating the Book

The book was created in plain text files, originally in ReStructuredText mode, written (and edited) on both Windows[2], or on Linux - which I use for all my personal and business needs at home.

These RST files were simply a quick way to gather notes. They were then later enhanced by adding more detail and/or example code, and converted to LaTeX by judicious use of the Pandoc text conversion utility, which I *highly* recommend.

The converted LaTeX files were then further enhanced, indexed, tidied up, etc using TexStudio which runs on both Windows and Linux, so I had the same development environment in both locations. Handy.

[1] I'm getting old, every day it seems. I am starting to forget things I knew - they are ageing out of the cache! I now appear to have the memory capacity of a small newt, so I need to have things written down!

[2] At work, in my lunch hour

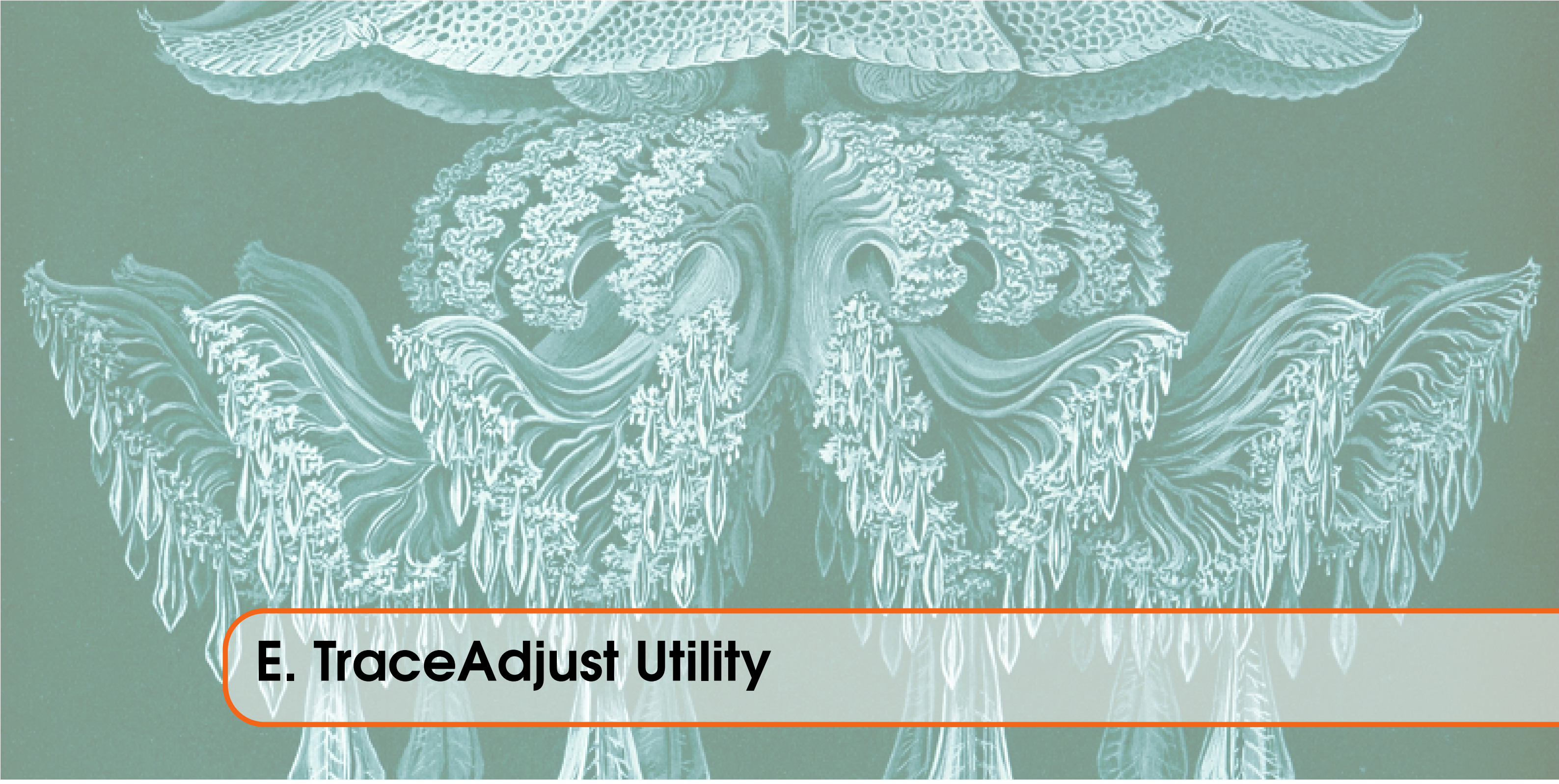
The book itself, was created using a LaTeX template called [the] Legrand Orange Book Template created by Mathias Legrand. It is thanks to him that you get this book for free, because the licence terms of the book template specify no commercial use. I'm happy with that myself.

The front cover image on this book is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Discomedusae* which are a taxonomic group of jellyfish.

You can read about them on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Discomedusae[3] have absolutely nothing to do with Oracle or Trace Files - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

[3] DiscoMedusae? Sounds like dancing jellyfish to me!



E. TraceAdjust Utility

TraceAdjust Introduction

TraceAdjust is a utility that I wrote to help me process the myriads of trace files that I come across in my DBA work. You can get the source code from github in my TraceAdjust repository and compile it on Windows or Linux/Unix with any decent C++ compiler.

It reads a normal trace file and writes out an adjusted one, as follows:

- The `tim` values are converted to seconds, by inserting a decimal point in the appropriate position;
- It adds a `delta` to each `tim` line. The delta is the number of microseconds between the `tim` on this line, and the `tim` on the previous (appropriate) trace line. This allows me to see how long passed between the previous `tim` and this one. Occasionally useful.
- It adds a `dslt` to each `tim` line. This is the "delta since last timestamp" and simply counts up the number of microseconds that have passed since the trace file last produced a timestamp line similar to the one in the header. Again, occasionally useful.
- It adds a `local` to each `tim` line. This is a conversion of the `tim` value on the line, to an actual date, in the current local timezone. The time part is resolved down to microsecond level. This is usually very useful!

Running a trace file through TraceAdjust will create a new trace file, which some trace analysing utilities cannot cope with due to the additional fields that I have introduced. The Trace File Browser, in Toad, on the other hand, copes with my trace files quite happily and simply ignores the additional data as appropriate.

The example below shows the before and after state of a `PARSE` line from a trace file:

```
PARSE #4474286416:c=0,e=418,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=4,plh=0,tim=1030574627220
```

Listing E.1: TraceAdjust Example - Before Processing

Which becomes:

```
PARSE #4474286416:c=0,e=418,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=4,plh=0,
    ⟹ tim=1030574.627220,delta=-1,dslt=768371,local='2017 Mar 13
    ⟹ 09:23:21.768371'
```

Listing E.2: TraceAdjust Example - After Processing

F. TraceMiner2 Utility

TraceMiner2 Introduction

So, you have a trace file, chock full of statements with lots of bind variables in use. You *need* to read through it to find out which execution of any of the statements used which actual bind values. How is this easily done?

TraceMiner2[1] is another utility that I wrote to help me process the myriads of trace files that I come across in my DBA work.

You can get the source code from github in my TraceMiner2 repository and compile it on Windows or Linux/Unix with any decent C++ compiler.

It reads a trace file and writes out an HTML report showing various details of the SQL in the file, showing:

- The line number of the file where the SQL statement was found at (`PARSING IN CURSOR`);
- The line number of the file where the `PARSE` statement was found at;
- The line number of the file where the `BINDS` details were found at;
- The line number of the file where the `EXEC` statement was found at;
- The depth (`dep`) of the statement when parsed etc.

You can choose to ignore statements over any given depth, so if you only want top-level and `dep=1` statements, just request `-depth=1` on the command line.

The default HTML report appears as follows:

[1] There will be no more unashamed plugs in this document, I promise.

TraceMiner2

Processing Trace File: test2.trc

EXEC Line	PARSE Line	BINDS# Line	SQL Line	DEP	SQL Text
44	27	28	25	1	select obj#,type#,ctime,mtime,stime, status, dataobj#, where owner#=90 and name='PK_RECONCILE_CAPSIL_UV' and r linkname is null and subname is null
60	53	54	51	1	select audit$,options from procedure$ where obj#=90851
76	69	70	67	1	select owner#,name,namespace,remoteowner,linkname,p_tir (property,0),subname,type#,d_attrs from dependency$ d, p_obj#=obj#(+) order by order#
98	91	92	89	1	select order#,columns,types from access$ where d_obj#=!
125	108	109	106	1	select /*+ index(idl_sb4$ i_idl_sb41) +*/ piece#,lengtl

However, you can, if you have a standard report format at your company, configure the generated css file to match that of your format. *TraceMiner2* will not overwrite the css file if one exists in the output folder.