Database Administration

by
Howard J. Rogers

Contents

Introduction

This document is the first in a series designed to provide a commentary on the Oracle 8i DBA Training Course slides. This particular one discusses only Chapter 1 of the course material.

The order of some of the slides in the official material is sometimes rather peculiar, and I've therefore told the story in the way and order that makes most sense to me, rather than stick rigidly to the slide numbering scheme employed by Oracle. That tends to be the way I do it in the training room in any case.

1.1 First Principles

An Oracle Database and an Oracle Instance are two completely different things.

- A Database is the set of physical files that reside on disk, which contain the tables and indexes and so on in which we are actually interested.

- An Instance is a structure in memory, together with its attendant processes, which manage the physical database.

Profoundly simply statements both, yet this is an important distinction to grasp. Databases live forever –they are comprised of disk files, after all. Instances die when the power to your server dies, or when your server suffers a Blue Screen of Death. They are transient structures, residing only in volatile RAM.

Some semantics flow from this distinction, too.

You cannot 'start up a database', for example. You **start** the Instance, and it will then proceed to open the database for us. Similarly, you **shutdown** an Instance, not a database: the Instance will (ordinarily) close the database for us.

An Instance can only ever manage one Database. That is true, even in exotic configurations, such as Parallel Server (where the one Database is managed by multiple Instances –each of those Instances can only be managing the one Database).

1.2 The Instance

Remember the earlier definition:

An Instance is a structure in memory, together with its attendant processes, which manage the physical database

Let's take that a stage at a time.

The 'structure in memory' is called the System Global Area (SGA).

The SGA is itself comprised of three main structures:

- The Shared Pool
- The Log Buffer
- The Data Buffer Cache

The Shared Pool contains two main parts: the Library Cache is where we store every piece of SQL that is ever issued against the database, together with the *execution plan* that describes how the database should deal with the SQL (where the data resides, whether there's an index that can be used to speed up access to the data, and so on). We store this in memory for each SQL statement because, if the same SQL statement is ever re-submitted by a User, we can simply re-use the execution plan… so things will be faster second time around. Then there's the Data Dictionary Cache, which stores details of what actually exists within the database (the tables, the indexes, the columns within a table, and so on), and the privileges each User possesses. We need that in memory so that we can, when a new SQL statement is submitted, rapidly check that it makes sense –that the table it wants data from actually exists, and that you have the right to see that data, for example.

The Log Buffer is used to store the Before and After Images of data that is subject to a piece of Data Manipulation Language (DML). DML means, basically, an update, insert or delete. If you are changing data, we store what it used to look like and what it now looks like in the Log Buffer. This allows us to re-perform (or "Redo", hence the name) each transaction in the event of some disaster befalling the database. Incidentally, you may wonder what the Before image of an Insert is… actually, it is practically nothing –just a pointer to where the new record is about to be inserted. Similarly, the After image of a delete statement is just a pointer to where the record *used* to be. One half of the entries into the Log Buffer in the event of either a Delete or an Insert will therefore be practically nothing – whilst the other half will be the entire record. If you do an update, however, the Before and After images are just the piece of data being modified –that is, the individual field or fields- not the entire record.

The Data Buffer Cache is where we actually store the data in memory that Users are requesting to see or modify. If someone asks to see what Bob's salary is, we will fetch that data up off disk, and load it into the Data Buffer Cache, and return the requested result from there. We then leave that data sitting in the Buffer Cache –on the off-chance that another User will want to know the same thing shortly thereafter. If the data they need is already sitting in memory, we won't have to do a slow disk fetch to get it second time around.

Be aware that we *never* simply go and fetch single records off disk… that would mean constant trips to the disk, and things would accordingly be extremely slow. Instead, we go and fetch the record *and the data around it*. How much data we actually fetch depends on each Database's configuration, as we'll see, but it's usually 8K these days. Older databases may fetch 2K-worth of data. This chunk of surrounding data is called an ***Oracle Block***, and the presence of *blocks* of data means that a request for a single column for a single record results in the entire block in which that record is found being loaded into the Data Buffer Cache.

I suppose the theory is that if you are presently interested in Bob's salary, it won't be too long before you ask after Sue's –and getting the both of their records in one hit is far more efficient than making two separate trips to disk.

So that's the SGA; what about the "attendant processes"?

There are five compulsory processes, without which no Instance can do its stuff. Because they run continuously in the background, they are generically known (imaginatively enough) as ***Background Processes***.

- Process Monitor (**PMON**) looks after the Users that connect to the Instance. If PMON notices that a User has abnormally terminated his connection to the Instance, perhaps because he simply switched off his PC, or because the network connection has failed, then it will find out what that User was doing, and tidy up the mess left behind. In particular, a User may have been in the middle of a transaction when the connection failed –that means record locks left lying all over the place, and partially-completed bits of DML that need to be reversed. PMON will free up those locks, and reverse (or 'rollback') the entire transaction.

- System Monitor (**SMON**) performs a number of roles, but it's main one is to check when we first open the Database that it is clean and consistent –and if it discovers the Database to be *in*consistent, it will initiate the necessary recovery steps for us. Sometimes, it will do *all* of the recovery required. Sometimes, it will need our intervention. But in either case, it is SMON that gets things underway.

- How does SMON know whether the database is clean and consistent? The answer is that the Checkpoint process (**CKPT**), from time to time during the normal operation of the database, will write a sequence number into the headers of every file associated with the Database. It is these numbers that SMON reads. If all files have the same numbers, we know the Database is consistent; if not, some sort of recovery is required.

- Log Writer (**LGWR**) is an important background process, that is forever writing the contents of the Log Buffer out to disk. Remember that the Log Buffer contains the Before and After images of data subject to DML, and these images allow us to re-perform all our DML in the event of disaster. Clearly, therefore, the Log Buffer contains crucial information –so crucial, in fact, that we would be mad to leave it floating around in volatile memory for very long. Hence, we need to save it to disk at fairly frequent intervals… which is what LGWR does for us, and what makes LGWR so important a process.

- Database Writer (DBWR) does much the same sort of thing, but for the contents of the Data Buffer Cache. Remember that the Buffer Cache is where blocks of data are loaded from off disk. If we are updating the data, or deleting it, or inserting new data, it is in the Buffer Cache that those changes are initially made. There must come a time when what has been changed in memory needs to be written back down to disk, and saved permanently….and indeed, that is exactly what DBWR does for us. I always think of DBWR rather like the 'File – Save' option found in your favourite word-processor …without it, your literary masterpieces wouldn't exactly last for long!

Now, there are plenty of other background processes which are important for the proper operation of production databases –in particular there is one called Archiver (**ARCH**) which we will meet in great detail later on. But these five are the crucial, compulsory ones for all Oracle installations.

1.3 The Database

So much for the Instance in memory –what about the physical Database on disk?

An Oracle database consists of just three groups of files:

- The **Control File** really acts as the keys to the Kingdom, for it contains (amongst other things) pointers to where all the *other* files in the database are stored. It is the first file read when we open our database, because we obviously need this crucial information before we can open and read anything else. Clearly, losing your Control File is a pretty serious matter –all your data and transaction information might be safe and sound, but we won't be able to find it –or, more accurately, the *Instance* won't be able to find it- because the pointers to the relevant files have been lost. For this reason, whilst it is true that a database only technically requires one Control File, it is usual to *multiplex* or mirror it, so that we have multiple, identical copies. Once that's done, you can lose one of the copies without losing too much sleep. You can of course mirror at the hardware level, but Oracle actually will perform the mirroring for you if you need it (and, believe me, you need it!)

- **Data Files** are where we store the stuff that most people would associate with a database –tables and indexes, containing the actual data we are interested in. Every Oracle database must have at least one data file –it's created for us automatically when we first create the database, and called the SYSTEM data file. Theoretically, it's possible to use this one data file for all our needs, provided you make it big enough. This would be, however, extremely *bad* DBA practice. When you update a table, there's almost certainly an update required to the index on that table… that's two simultaneous updates. If both the table and the index are sitting in the one data file, the two updates have to queue up behind each other, which means slow performance. Much better to have tables in one file, and indexes in another –and house the two on separate physical hard disks. That way, the two updates can take place simultaneously, and performance improves hugely. For this reason, at least 3 data files would be useful -and other factors we'll discuss later mean, indeed, that it is usual for most Oracle databases to end up with at least six data files –you have to manually add in the five additional ones yourself. That said, be aware that Oracle can actually support up to 65,000+ data files. And whilst this is a bit extreme, many databases will commonly have tens, to a couple of hundred, of data files.

  It is to the data files that DBWR flushes the contents of the Data Buffer Cache. It's also where the blocks of data are read from when we load data *into* the Buffer Cache.

- The Before and After images of data that has been subject to a DML statement are stored, initially, in the Log Buffer. But, as we've already discussed, it's apparent that this very important data should eventually find a permanent home on disk –and the Redo Logs are that permanent home. In other words, when LGWR decides to write the contents of the Log Buffer to disk, it is to the Redo Logs that it turns. There must always be at least two Redo Logs. That's because when we fill up one file, we need to be able to keep writing information somewhere –otherwise, we would have to stop any further DML statements (which translates to having to engineer a system hang, which is clearly not a good idea).

  Still, what happens when the second log fills up, too? Do we have to have an infinity of Redo Logs to prevent the system hanging? No – because what we do is simply switch back to the first log, and over-write the contents with the fresh Redo information. In this way, two logs will just about suffice for a database: we continually switch between them, over-writing whatever we find there already, in a never-ending cycle. (It's rather more usual to have three or even four Redo Logs, but two is the minimum).

What I've just described is the default behaviour for Oracle –go right ahead and over-write the contents of the Logs. But you will recall that I've also described the information that we write to the Logs as "crucial" –are we sure we want to be over-writing the information that allows us to re-perform transactions in the event of disaster? In many production the databases, the answer to that is clearly going to be a big, unequivocal "No". So Oracle has a mechanism whereby the contents of each Redo Log can be copied out to another file –and, once we have that copy safe and secure, we can merrily over-write the Redo Log without a second thought. This copy is called an 'archive redo log', and archiving makes all the difference between being able to totally recover a database, and having to sign up to lost data by restoring everything from last night's backup. Archiving is a big backup and recovery issue, and I'll deal with it at length later. For now, note that the possible presence of archived redo logs means that we ought to get into the habit of distinguishing between Archived Redo Logs and their On-line Redo Log cousins. The 'live' ones, that LGWR writes to, are the "On-line Redo Logs".

So, that's all there is to an Oracle Database: Control Files, Data Files and On-line Redo Logs.

- Clearly, however, there are other files floating around the place that are very important –**Archived Redo Logs**, as we've just discussed, for example. As I've mentioned, these are of supreme importance for total database recovery. Nevertheless, they are *not* technically part of the database –for the very simple reason that once Oracle has created each Archive, it doesn't keep track of it. If Junior DBA walks in one day and deletes the lot, Oracle won't even notice the loss, and certainly won't complain about it. (You, on the other hand, will most certainly complain about it when you are trying to recover a critically important database and find that you can't do so because of missing archives!).

- Similarly, the **Password File** is a fairly important physical file that contains the details of which people are allowed to perform *Privileged Actions* on the database. A Privileged Action is basically one of five things: starting up the database, shutting it down, backing it up, recovering it in the event of disaster and creating it in the first place. If you want to do any one of these five things, you must be authenticated as a special, Super-Hero, kind of User –and therefore have an entry in the Password File. Clearly, then, the Password File is important for the successful management of a database –but it isn't technically part of the database, because, again, Oracle won't notice the loss of the file until you try and use it one day. It's not keeping constant watch on it as it does with the Control Files, Data Files or On-line Redo Logs (remember that with these files, CKPT is periodically updating the file headers with a sequence number, and will throw a big fit if it fails to find a file it's expecting to update. Definitely not the case with the Password File).

- One other physical file that is not part of the database ought to be mentioned –because without it, you can't actually start the Instance or open the Database in the first place! Nevertheless, it's also not technically part of the database. This is the **Parameter File**, or more commonly (and inaccurately) known as the "init.ora".

  The Parameter File is just a text file (the Password File and Archived Redo Logs are binary encoded). It contains a stack of keywords which govern the way in which the Instance and Database work. For example: if you're going to build an SGA in memory, wouldn't it be a good idea to know exactly how big it should be? Absolutely: which is why the Parameter File has the keywords "SHARED_POOL_SIZE", "LOG_BUFFER" and "DB_BLOCK_BUFFERS" –you set these to a given number of bytes, and when you issue the 'startup' command, Oracle will construct an SGA sized according to these parameters.

  Similarly, by setting CONTROL_FILES to the relevant directory and filename combination, you are able to tell the Instance where to find the Control File… remember that once we've got hold of the Control File, we then know where all the other files in the Database are to be found.

  There are around 200 parameters which can be set in the Parameter File –but the vast bulk of them need only be worried about when trying to perform fairly advanced stuff (like configuring Parallel Server installations, for example). The four I've just mentioned are perhaps the commonest, together with one called DB_BLOCK_SIZE, which determines the size of an Oracle block –in other words, how much data should be read off disk each time we have to visit it.

  If you don't actually go to the effort of setting these various parameters, there are usually default values which Oracle will supply for you (not in all cases, however). Be warned, however, that Oracle defaults are not renowned for always being the most sensible values in the world –and a good deal of the art of tuning a database properly lies in determining more appropriate values for a fair few parameters.

1.4 Connecting to an Instance

Users never connect directly to an Instance. That is, it is not possible to fire up a program on a PC and somehow poke around inside the Data Buffer Cache to see what Bob's salary is currently set to.

Instead, Users fire up programs which generate **User Processes**, and the User Process in turn causes the Server to spawn a **Server Process** -and it's the Server Process which does all the fetching and updating and inserting of data on the User's behalf.

For many databases out in the real world, the relationship between a User Process and a Server Process is strictly one-to-one: each new User wanting something out of the database will generate a unique User Process, and will communicate exclusively with their own Server Process. No other User can share a Server Process. This sort of configuration is accordingly called "Dedicated Server Configuration", and it's the commonest way of configuring Oracle.

It is possible to configure Oracle for what's called Multi-Threaded Server (MTS) operation, however, and when run like this, Server Processes *are* shared amongst Users. It's a fairly exotic configuration, however, and won't be further discussed in these notes.

Quite frequently, Users will request that data extracted out of the database should be sorted or grouped in some way before they see it –for example, 'select * from employees order by department' will require that all the employee records be pre-sorted by the department field before being returned to the User. It is the Server Process again which is involved in this sorting process, and it performs the actual sort in a special area of private memory called the **Program Global Area** (PGA).

The size of the PGA available to each and every Server Process is determined by a Parameter File setting, SORT_AREA_SIZE (note that the name of the parameter is a constant reminder of the primary function of the PGA). Since the Parameter File only has one such setting, it should be fairly obvious that ALL Server Processes will have access to the same size PGA – you can't have some Server Processes with more PGA than others, and that basically means that all Users are considered equal as far as sorting is concerned!

That pretty well completes the description of the Oracle architecture: An Instance comprised of an SGA and background processes, managing a database comprised of Control Files, Data Files and Redo Logs -with a few extra files thrown in to make things work, and Users connecting to the whole edifice via Server Processes.

1.5 Naming the Instance and the Database

Before discussing how the architecture actually works when it comes to performing queries and DML, two fairly important points need to be made: Instances have names, and so do Databases.

We have to have a unique name for an Instance, because it's perfectly possible to have multiple Instances running on the one Server, and we need to know at any one time which one we are connected to, or which one is causing us grief.

The same holds true for Databases –the only thing stopping you having a hundred databases on the one server is the amount of disk space they would consume! So we need to uniquely identify each Database, too.

Notice that these are two separate considerations: the Instance name is totally independent of the Database's, and they don't have to be the same (though for ease of management, you'd be much better off making them so).

So how do you name an Instance? Simply with an environment variable, set using whatever command your particular operating system dictates. The variable name is always ORACLE_SID. You set that before trying to start up, then when you actually issue the startup command, the Instance that is built acquires the current ORACLE_SID as its name thereafter. If you then want to start up a second Instance, you simply set ORACLE_SID to be something else, and issue yet another startup command.

This same technique works when it's a question of *connecting* to different Instances, rather than creating them in the first place. For example, if I set ORACLE_SID=ACCTS, and then issue the connect command from within SQL*Plus, I will connect to the Instance previously created with the ACCTS name. If I then want to switch to the SALES Instance, I just quit SQL*Plus, re-set my ORACLE_SID, fire up SQL*Plus once more, and issue a new connect command. Oracle connects to the Instance with the same name as your current ORACLE_SID.

On NT Platforms, you set the ORACLE_SID variable simply by typing 'set ORACLE_SID=XXXX' in a DOS Window. On many Unixes, it's all a question of typing 'export ORACLE_SID=XXX'.

What about the name of the Database? That's actually set when you first create the database, and is stored in the Control File (so it can't easily be changed). There's a parameter in the Parameter File called db_name which is involved in the naming of the Database, although the thing that actually clinches it for all time is part of the 'create database' SQL syntax, which we'll cover later in these notes.

Notice that there is ample scope for confusion with this naming scheme: the ORACLE_SID is independent of the Database name, and can easily be changed, whereas the Database name can't. Pursuing the earlier example, there is nothing to stop us setting ORACLE_SID to SALES, and then opening the ACCTS database with it, or vice versa… and things will function perfectly normally, but everyone will end up with a big headache working out what is going on! Tip: make SID equal the database name, and make sure the right SID is set when you are starting up Instances.

1.6 Performing Queries

A query is simply any form of select statement –you simply want to see data of some sort, not do anything to it. Queries therefore do not modify data, and accordingly they don't generate Redo. They do, however, return results to Users –you get to see records scrolling up your screen.

Every query goes through three different phases of processing.

- The **Parse Phase** involves Oracle checking the basic syntax of the select statement you've submitted (did you type "select" or "solect", for example?). Assuming your request passes that test, a check is made for an exactly matching query in the Library Cache. If there is a matching query there, we can skip the next few steps, and thus save ourselves a fair bit of time and effort.

  Assuming this is a brand new query without a Library Cache match, we need to check that it is asking for sensible information –are you selecting from a table that actually exists? Are the column names you used correct? Finally, we need to know that you, as a User, actually have the requisite privileges to see the data you are requesting.

  To do all those checks, your Server Process makes trips to the Data Dictionary Cache in the Shared Pool. Remember that the Dictionary Cache tells us what objects exist in the database, what columns exist within tables, what Users have which privileges, and so on. Whilst visiting the Dictionary Cache, the Server Process will also discover whether there are any indexes on the table you are interested in –and whether using these would speed up the data retrieval process.

  At the end of the parse phase, we therefore know that what you are querying exists, that you have rights to see it, and the best way of getting at that data. Your Server Process summarises all these findings in an **Execution Plan**.

- We then move to the **Execution Phase** of query processing. In this phase, we simply carry out whatever the execution plan tells us to do. The Server Process will first check in the Data Buffer Cache to see if any of the requested data is already in memory –if it is, we needn't drop down to the disk to retrieve it. If it isn't, the Server Process *will* go to disk, read the relevant Oracle blocks and load them into the Buffer Cache. If you've requested a lot of data, we keep reading as many blocks up into the Buffer Cache as are needed.

  The execution phase ends when all the requested data is memory, in the Data Buffer Cache.

- Finally, we move into the **Fetch Phase**. This simply means that the Server Process starts returning the appropriate data blocks back to the User process –at which point, you start seeing the results of your query appearing on your screen.

  If your query had requested that the data be grouped or sorted, the Server Process would do the necessary processing of the data in its PGA before showing you the results. If the PGA runs out of space, the Server Process would swap the PGA's contents down to disk (usually to the "TEMP" data file), free up the PGA for further work, and keep going. Many swaps to disk might be needed for a large report. The Server Process then reads the mini-sorts back off disk, and merges them together to produce the final report.

Note that all queries have a parse, execute and fetch phase, and that it is the Server Process that does all the work during each of these phases.

The only real subtlety that arises when performing queries is the one I mentioned earlier during the discussion about the parse phase: can we avoid all the lengthy business of checking for table names, column names, the presence of indexes and so on by simply re-using an identical query previously submitted and stored in the Library Cache?

To determine that two queries are identical, Oracle hashes their text into a unique identifier… if the two identifiers are equal, then the text must be equal, and therefore the execution plan for the first one, already calculated and raring to go, will do perfectly well for the second.

However, for this to work, the text of the two queries must be *absolutely* identical. These two will not pass muster:

select * from emp;   and
select * from Emp;

One refers to the emp table with a lower-case "E", the other uses upper case –that's sufficient for the two statements to hash to different identifiers, and hence be regarded as different queries.

For this reason, a consistent use of SQL by developers is most important: if one half of your development team upper-cases all table names, and the other doesn't, then your resulting application will be doing far more parsing of SQL statements than is really necessary, and as a result your application will run appreciably slower than it otherwise would.

Somewhat outside of scope as far as these notes are concerned, but the use of 'bind variables' is equally effective at cutting down unnecessary parses and execution plan generation.

1.7 Performing DML

There are three kinds of DML: Inserts, Updates and Deletes. The Oracle architecture deals with each sort of DML statement in exactly the same way. For the sake of simplicity, I'll assume we're going to do a simple update on a record in the emp table –just bear in mind that what follows would equally apply if I had decided to insert a new emp record, or delete an old one.

The first thing to say about DML statements is that they don't generate feedback at your terminal: you don't, for example, get a listing of the new records you've just inserted. In slightly more technical terms, that means that there is no Fetch phase for DML. If you want to see the results of your inserts, updates or deletes, you have to issue a second, separate query.

Nevertheless, DML statements do have a parse phase, and they do have an execution phase, too. What takes place in these two phases is exactly as I have described earlier for queries, so I won't repeat it here in full: suffice it to say that we check the syntax of the statement, check that it references actual objects and columns contained within the database, and check that you have the necessary permissions to do what you are seeking to do. We then retrieve the data off disk (unless it is already in memory, of course).

You might well ask what we retrieve off disk if we are doing an insert -since clearly the record cannot already be on disk. In fact, we retrieve the block where the new record will reside. That might be a completely empty block, or a partially-filled one with enough room to house the new record. So even for an insert, there is still the need to obtain an appropriate block, and hence (perhaps) to visit the disk.

Once the block is in memory, the interesting stuff starts. There are basically 4 steps involved in carrying out the DML statement:

- **Lock the Record**: Locking in Oracle is at the record level, so the fact that I am updating Bob's salary won't stop you doing things to Sue's record in the same block. However, it is the **entire** record that is locked, so whilst I am changing Bob's salary, you won't be able to alter his bank account details (for example).

- **Generate Redo**: We need to be able to repeat this piece of DML in the event of disaster. In order to pull off that particular trick, we need to know what the data looked like both before and after it was altered by the DML. Accordingly, we generate a before image and an after image of the data, and write that into the Log Buffer.

  When the DML is merely updating an existing record, both the before image and the after image will be very small: in fact, just the field being updated, and a small record pointer indicating where the change is being made. For example, if we were updating Bob's salary from $800 to $900, then these two figures (plus the record locator) would be the only thing written to the Log Buffer.

  When the DML is either an insert or a delete, then at least one half of the redo information written to the Log Buffer is the entire record. The before image of a delete is the entire record; the after image of an insert is, likewise, the entire record. In both cases, too, the other half of the redo information is just the record locator (i.e., where will this record be deleted from, or where will it be inserted). Note that, in either case, considerably more redo is generated than would be by a typical update.

- **Generate Rollback**: We need to give you the opportunity of changing your mind, and un-doing the transaction. To this end, we need to write the before image of the data into a new buffer in the Buffer Cache, called the "rollback block". The same rules as to what constitutes the before image of the data applies to rollback as it does to redo.

  Once the rollback block is obtained and filled with the appropriate before image data, the lock that we took on our row at step 1 is updated so that it points to the rollback block. This is significant in that it is the *lock* that knows where to retrieve data for rollback. Once that lock is removed, therefore, we won't know where to find the rollback data, and rollback will therefore not be possible –even if the rollback block is still floating around in the Buffer Cache. It might be there –but we won't know where to find it!

- **Change Data**: Now that we have the ability to re-perform the transaction, and the ability to change our minds about it, we can go ahead and actually carry out the requested modification to the data. Changing the data does not remove the record lock: Bob would be sitting there at $900, for example, at this stage, but his record is still locked until the update is committed.

I might point out here that all of the above steps are carried out by your Server Process. It's only when we get to the commit stage that other parts of the Oracle architecture really come into play.

- **Commit**: To make any DML permanent, you must issue a separate 'commit' request. When you do this, your Server Process writes a System Change Number (SCN) into the Log Buffer to indicate that the transaction has been committed. LGWR is then prompted into action, and it flushes to the current online Redo Log everything in the Log Buffer not previously flushed. Once LGWR has finished writing, your Server Process returns a 'Statement Processed' message to your User Process –in other words, you see confirmation on screen that the record has been committed. Finally, the lock previously taken on the record is removed (and hence we lose the ability to rollback the transaction, as described earlier).

Note that a commit in Oracle does *not* result in the changed data itself being written back to the data file. It is only dependent on LGWR successfully writing the redo information to the Redo Log. There is a very good reason for this: it's much quicker this way round. If a commit depended on writing buffers out to data files, we would have to wait for DBWR to write the entire block to disk (recall that the Oracle block is the smallest unit of i/o as far as data is concerned) –we're talking about perhaps 8K or more of writing to complete. LGWR in contrast only has to write what's in the Log Buffer –which, as the earlier discussion explained, is tiny byte-sized stuff for the average update, and only the length of the entire record for the average insert or delete. For this reason, Oracle calls this commit mechanism a 'Fast Commit'.

Is it safe? Absolutely. It does mean that the changed data is floating around in memory without being saved to disk, and is thus vulnerable to server failure. If the power to your server suddenly failed, you would indeed lose the buffer containing the changed data… but because the before and after images are safe in the Redo Logs, we have all the information needed to re-perform the transaction, and when power is restored to the server, that's exactly what happens. So the changed data is *recoverable*, and hence it is perfectly true to say that committed transactions are not lost.

When will the changed data actually be written back to the data file? When DBWR feels like it is the short answer. Various things cause DBWR to spring into action:

- When a checkpoint is issued (shutting the database down cleanly will issue one, for example)
- Someone needs stacks of blocks for some reason, and there aren't enough free ones available (DBWR flushes out old ones to make room)
- If the number of dirty blocks reaches the threshold set by the DB_BLOCK_MAX_DIRTY_TARGET parameter set in the init.ora
- Every three seconds, by default.

That might sound quite frequent (especially the last point), but in reality it's not. Three seconds in an OLTP environment is an age, for example. The point is that you really don't *want* DBWR working too hard, because it would mean a lot of i/o taking place, and potentially slower performance as a result. And, provided our Redo Logs are safe, we don't *need* DBWR working hard… the data is safe in any event.

Whilst we're at it, we might as well talk about when LGWR is prompted to write down to the Redo Logs. We've already mentioned one of them –at every 'commit', but the complete list is:

- At a commit
- When the Log Buffer gets a third full
- If there's 1Mb of un-flushed redo
- Before DBWR flushes to disk

The Log Buffer gets reused in a cyclical sort of fashion –so when it's completely full, we simply start overwriting the stuff at its beginning. Before overwriting such information, we must clearly make sure its safe on disk (otherwise we'd lose it) –but if we left it to the end of the Buffer before flushing, the system would have to grind to a halt whilst a massive flush took place. Clearly not a good way to run things –so Oracle instead flushes as each third of the Buffer is reached. Similarly, if your Log Buffer was, say, 6Mb big (a very unusually large Buffer it has to be said), then every third would represent 2Mb of Redo –and that could take a while to flush. So instead, LGWR kicks in every 1Mb if we ever get that far (obviously, if your Log Buffer is a more reasonable 500K in size, we'll never reach this particular trigger point).

The last thing that causes LGWR to flush is perhaps the most significant. We might be nowhere near 1/3rd full, nor at the 1Mb trigger point –but DBWR suddenly decides that the Buffer Cache has run out of room, and therefore needs to be flushed to the data files. If we allowed DBWR to flush *before* LGWR, and then had a catastrophic power failure, we will be in a complete mess: there will be data sitting on disk that was in the middle of being updated by a transaction, and data that was appropriately committed –but we won't be able to tell the one from the other, because it's the Redo information (specifically, the SCN number) that tells us that. It's thus more important to get LGWR to flush successfully than to have DBWR do its stuff. Hence, any time DBWR wants to write, we always fire off LGWR first.

1.8 Some Corollaries

The previous discussion about how Oracle handles DML statements has some (perhaps) startling side effects which may or may not be obvious.

- *When normally operating, Data Files contain both committed and uncommitted data.*

Remember that DBWR will flush blocks from memory to disk according to its own schedule, and that schedule has nothing to do with Users issuing commit requests. That means I can be in the middle of a transaction, and DBWR might *still* decide it needs to flush the block(s) I'm working on out to disk. Hence uncommitted data gets written down to disk. Is this a problem? No –provided that the Redo Logs are safe, because the presence in the Logs of SCN information tells us everything we need to know about what stuff in the Data Files is committed or uncommitted. We can always sort it out, therefore –provided nothing happens to our Redo Logs!

- *When LGWR flushes to the Redo Logs it doesn't distinguish between committed and uncommitted transactions.*

If I'm in the middle of changing Bob's salary, and you issue a commit for a change to Sue's record, then your commit will cause LGWR to flush to disk – and it will write my transaction to the Redo Logs along with yours. That's because sorting out the uncommitted transactions from the committed ones would take time –and for performance's sake, we want LGWR to get the job over and done with as quickly as possible. Does it matter? No –because your transaction will have an SCN assigned to it, and mine won't. In the event of a disaster, we can then sort out the difference between the two, and correctly recover just the committed transactions

- *Recovery is a two-stage process*

When we have to recover the database, this sorting out of the committed transactions from the uncommitted ones has to take place. What that means is that, to start with, we re-perform *all* transactions regardless (this is called the **roll forward phase** of recovery). When all transactions have been rolled forward, we then proceed to identify all transactions which don't have an SCN assigned to them: clearly these are uncommitted transactions. We then roll those transactions back again (and this is therefore called the **rollback phase** of recovery). The end result is that transactions which were committed before disaster struck are correctly recovered; transactions which were pending at the time of the disaster will have been lost.

Because the rollback phase could take a long time, we actually open the database for business at the end of the roll forward phase… as Users request data, their Server Process will spot any dirty data in the blocks they're touching, and perform the rollback for that block before actually carrying out the User's original request. This means the database is open to Users more quickly than otherwise –but it also means that it responds more sluggishly for some time after opening than would be normal (all those Server Processes taking time out to do rollback when normally they'd just be servicing User requests).

Any blocks which aren't touched by Users within a reasonable amount of time will be rolled back automatically by SMON.

This principle of performing the rollback phase in the background after opening the database is called a 'warm restart'. The sluggish performance that results is the price we pay for not discriminating between committed and uncommitted transactions when flushing to the Redo Logs in the first place (but think about it: which would you rather suffer –constantly slow performance because LGWR as part of its normal job has to think about what to flush, or a one-off bout of sluggish performance after a database failure? How often are you planning on having your database fail?!)

1.9 Read Consistency issues

We have an inviolable rule in Oracle, sometimes expressed in the phrase, *"Readers never block writers, and writers never block readers"*. What that means is that the fact that I've selected to see Bob's salary shouldn't stop you from getting in there with an update to his salary. Similarly, if I'm in the middle of updating Bob's salary, that shouldn't stop you from being able to read his details.

Nevertheless, this behaviour raises an interesting question: if I'm in the middle of changing Bob's salary from $800 to $900, what figure for his salary should you see when you suddenly decide to query it?

In fact, you will see the old figure of $800 –in Oracle, you are never allowed to see values which have not been committed, unless you happen to be the person doing the updating in the first place.

But Bob's record really is sitting there with a salary of $900 –so how do we arrange for you to see the old figure of $800? The answer is that Oracle prepares for you a '**read consistent image**' of the data –i.e., an image of the data as it was before the pending transaction altered it.

What happens is that your Server Process notices the lock on Bob's record –and notices that it doesn't own it, but that someone else (i.e., me) does. Immediately, this prompts it to take an entire copy of the block in which Bob's record is stored. That copy includes the lock information –and you'll recall from our earlier discussion that the lock information contains a pointer to the rollback block, in which Bob's old salary is stored as part of my transaction. It's then a fairly simple matter for your Server Process to locate the rollback block, retrieve Bob's old salary, and roll back the data contained in the block copy previously taken. The data for your query is then fetched from this block copy (technically called the **read consistent block**), rather than from the 'real' block (technically called the **current block**).

This behaviour is apparently unique to Oracle –other database have the concept of 'dirty reads', in which you can force the fetch to take place from the current block. Unless you force such a dirty read, these other databases will ignore Bob's record altogether! Not in Oracle: you will always get all records, but you can never see the uncommitted values.

In fact, the only person who can ever see the uncommitted values is the one doing the transaction –if in the middle of my change to Bob's salary I decide to do a 'select * from emp', then my Server Process will notice the lock on Bob's record, but also note that it actually owns the lock –and thus the lock can safely be ignored for reporting purposes. This behaviour is fairly important: if I can't see what my changes look like, how can I be expected to safely commit them? Note, though, that it's the identity of the Server Process that allows this behaviour, not the identity of the User involved. In other words, if I'm changing Bob's salary in one session, and querying it another, I have two Server Processes –and the second one will be just as much not the owner of the lock as your Server Process would be. I won't be able to see Bob's new salary in that second session, therefore.