Oracle DBA 101:

A Beginner's Guide

Rao R. Uppaluri

Oracle DBA 101: A Beginner's Guide

ISBN: 1-58112-764-2

Universal Publishers/uPUBLISH.com
2000

www.uPUBLISH.com/books/uppaluri.htm

Preface

After simmering in the dynamic waters of research in Chemistry and Physics for nearly four decades, I first entered the field of business software as a hobby. This hobby turned into passion and took me to several areas such as Oracle 7.x, Oracle DBA, Developer 2000, Visual Basic 4.0 and Visual basic 5.0. I found that most of the books were written by stalwarts who were mathematically oriented and hence treated the topics in a dry fashion. Hence I had to read each topic myriad times to make sense. The present monograph on Oracle DBA is a treat of the topic in “English”, with examples from day-to-day experience and is ideal for newcomers to this field. This may be an oversimplified approach. In the process of simplification, rigour is the first casuality. However, I realise how difficult it is to discover examples that truly resemble any real-life situation. I have tried to give such examples that enable the reader to grasp the concepts.

This book is organised as follows. Chapter 1 discusses Oracle database in general and different components that go to make an *Instance*. Chapter 2 describes the estimation of the size of *Non-clustered* and *Clustered Tables*. Chapters 3 describes routine monitoring including *Tablespace Allocations, Freespace, Fragmentation* along with some related *Views*. In Chapter 4, preventive measures to avoid problems in the database are discussed. Chapter 5 includes a very important aspect of Oracle, viz., *Security*. In view of its paramount importance, a more detailed discussion is given exclusively for *Rollback Segments* in Chapter 6. Several aspects of *Backup* and *Recovery* and comparison of different *Backup* processes are presented in Chapter 7. Chapter 8 gives a flavour of *Audit* in Oracle. The concluding chapter provides examples of minor changes in the code of application programs that result in optimal utilisation of resources.

As newer versions of Oracle become available, the need to write scripts for Database Tuning etc will not be necessary since they will come in packages. However, what these packages do inside the “box” will become clear after going through this book. Hence this book will be an useful tool at the introductory level for the Teacher as well as the Taught in Oracle classes.

I scrupulously avoided the usage of words like “Table” and phrases such as “For instance” in the conventional sense since these have different connotations in Oracle. Instead, I used “Illustration” and “For example”. All English terms not used in the conventional sense as well as Oracle keywords are italicised.

I believe that after going through this elementary monograph, reading other books by software giants like Kevin Loney, J B Greene and M R Ault may not be that initimidating. These authors have greatly influenced my thinking on this topic.

I would like to thank several people who made the writing of this book possible. Renuka Uppaluri my daughter, always had tremendous confidence in my capabilities, which I some times felt, was probably not justified. Even today, she feels the same. Without her encouragement, I would never have even entered the field of software. My other daughter, Prasanti Uppaluri feels equally about my capabilities but strongly feels that at this age, I deserve to rest rather than maintain this arduous schedule; but she never comes in my way. My wife, Vardhani Uppaluri like any house-maker, is neutral. Neither she has confidence in me nor is disappointed by me. But I can not imagine my life without all three.

Seelam Venkat Reddy one of my associates during Oracle/Developer course was my constant companion in developing an application program on Integrated Bank Branch Management. He was an exceptional student, not in a hurry to cross the seas until he had learnt enough. During this development, we learnt many tricks in Oracle, the hard way. It was a great experience. This gave a fillup for future endeavours including this monograph.

I have had the pleasure of learning the Oracle/Developer Course from M/S Syspro (XL Computers Ltd), Bombay, India, under the management of Mukul Gupta. He was kind enough to allow unlimited access for practicals as part of the course. I benefited tremendously from this facility. In addition, he also provided a facility for Oracle DBA practicals for a small fee. This helped a lot in understanding the topic.

Two practicing Oracle DBAs, Srinivas Sunkara, NC, USA and Ajit Nadgir, NJ, USA, kindly read the manuscript and provided a few comments. Renuka Uppaluri has painstakingly proof-read the manuscript for spelling and grammatical mistakes. Havish Koorapaty, my son-in-law, helped with the formatting of this book and I am thankful for his patience in performing this task that required several iterations.

Most of the material and ideas in this booklet originated from the Oracle Corporation's materials and books by the authors Ault, Greene and Loney. I added my simplified interpretation to these. I am indebted to the Corporation and these authors for the same. Oracle is a registered trademark of Oracle Corporation.

I am responsible for all omissions and commissions in the interpretation of the topic and in the manuscript.

I shall be grateful to any reader who can provide critical comments and suggestions which will go in improving this book in serving the purpose for which it was written, viz., simplification of the topic. My email address is ruppaluri@hotmail.com. Finally, I would like to thank Universal Publishers/uPUBLISH.com for accepting this book for publication.

Rao R Uppaluri
May-October 1999
Milwaukee,WI and Cary, NC, USA

Table of Contents

List of Illustrations

Chapter 1 Oracle Database

An Oracle database is a set of data. This set is stored and is accessed in a manner consistent with Relational Model. By database, one refers to a physical database along with physical objects, memory objects, and process objects. Data in a database is stored in *Tables*. Relational *Table*s are defined by their columns. An Oracle database stores its data in *Files*. Internally they are database structures that provide a logical mapping of data to files, allowing different data to be stored separately. These logical divisions are called *Tablespaces*. A database can have several *Tablespaces* as depicted in Illustration 1.1.

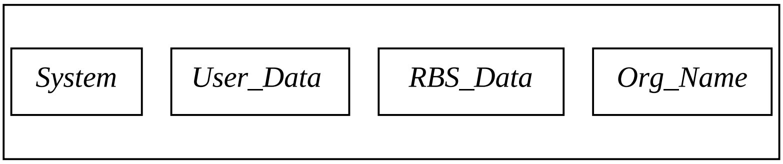

Illustration 1.1 - My Database's Tablespaces

In a typical database I dealt with, the *Tablespaces* are named as *Org_Name* (Name of organisation), *System, User_Data, Temporary_Data, RBS_Data* etc. Each *Tablespace* has a specific purpose. For example, *System Tablespace* contains what is called a *Data_Dictionary*, a sort of an information centre about the data, data type, length, precession and database objects (e.g., *Tables, Indexes, Synonyms, Views, Grants, Roles* etc.). Thus, *Data_Dictionary* gives almost complete information about all aspects of the database content. *Data_Dictionary* is an important source of information about the database and hence its contents should not be accessible to all. It is like an inventory list of items in a jeweller's shop.

A *Tablespace* is a logical division of a database. A given database must have at least one *Tablespace* to start with. This essential *Tablespace* is known as *System Tablespace* (like a Government Office) from where the database is run. A *Tablespace* can belong to one and only one database. In Illustration 1.1, *User_Data* is a *Tablespace* which is used to contain several *User's* data. *RBS_Data* is yet another *Tablespace*.

Each *Tablespace* contains one or more *Files* called *Datafiles* (e.g., *Table*). A given *Datafile* (also called *Segment*) can belong to one and only one *Tablespace*. *Datafiles* are fixed in size at the time of creation. As the data grows in the file, and more space is needed, new files are added. The different possible segments are: *Table Segment*, *Index Segment*, *Rollback Segment* and *Temporary Segment*.

Dividing database objects among multiple *Tablespaces* allows these objects to be physically stored in separate *Datafiles*, which are placed on separate disks. For example, *Data_Dictionary Table*s which provide a catalogue, are used by the *System* to manage itself. All *Data_Dictionary Tables* are stored in the *System Tablespace* and are accessed by one *User*, *Sys* (equivalent to a Government Official). *User Sys* owns the *Data_Dictionary Table*s. The *User System* owns the *Views* that access these *Data_Dictionary Table*s for use by the rest of the *Users* in the database. It may be recapitulated that a *View* is a *Selection* of one or more rows from one or more *Tables*. The data is not duplicated. This means that the *View* does not occupy any storage space in the database. The *View* can neither be altered nor can any *Indexes* be *created* on any columns of the *View*. The definition of *View* (including the query that *created* it, its column lay out, and the *Privileges granted* on it) is stored in the *Data_Dictionary* of the database.

In my database, *Datafiles* of all *Users* are stored in different folders in the *Tablespace, User_Data*. Any database internally comprises *Schema, Tables, Users, Indexes, Clusters, Hash-clusters, Views, Sequences, Procedures, Packages, Triggers, Synonyms, Privileges, Roles, DB Links, RBS* etc. Since these are relatively simple and known to all the Oracle *Users*, they are not dealt with here. *Privileges and Roles* are discussed later (Sections 5.2 and 5.5).

In addition to the above items, the database also contains internally two memory structures known as global area and process area. The global area comprises *SGA* which consists of *Db_Block* buffer, *Dictionary_Cache*, *Redolog* buffers, *Shared_SQL_Pool*. The process area comprises the database's own background processes, managed by the database itself. These process structures are *SMON*, *PMON*, *LGWR*, *CKPT*, *ARCH* etc. These require a little exposition.

1.1 Memory Structures

As was mentioned before, there are two kinds of memory structures available in Oracle. They are global area and process area**.** The most important of these is the *System Global Area* (SGA). This is a composite of different sub memory areas within as indicated in Illustration 1.2.

Db_Block buffer	Redolog buffer	Dictionary_ Cache	Shared_SQL_ Pool

Illustration 1.2 - Structure of SGA in Oracle 7

The *SGA* is like a bulletin board. Its use can be understood as follows. The most efficient way of communicating information to all employees in an organisation is to exhibit the information on the notice board. As and when new information is to be communicated, old information is removed to make space for the new information. The memory areas are like the bulletin board. Let us see what each of these sub memory areas is for.

Db_Block buffer, as the name suggests, is a temporary (buffer) memory location in the *SGA* to hold the data blocks read by the server process from the segments (such as *Tables*, *Indexes*) of the database through an SQL query. This buffer occupies roughly 30%

of the *SGA* size. The size of this buffer is fixed and can be changed at will, if need arises. Since in a query such as:

```
SQL> Select * from Tn where parameter1 = m;
```

only a part of the *Segment* (*Table Tn*) of the database is read, the size of this buffer is smaller than the size of a database segment. So this buffer can not continue to keep all the data brought into it by all the queries. Like the bulletin board above, all old information is to be selectively wiped out to make room for the recent information. The buffer uses what is called Least Recently Used (**LRU**) algorithm to wipe out old information to make space for the new information.

We all use this LRU algorithm in every day life unknowingly. Suppose there is space for only two photo albums in the coffee table of your living room. When you are just married, you keep the engagement album and the wedding album. These are the most currently referred to albums by your visitors. After a few years, a child is born and a new album of the child gets filled and it takes the prime place on the coffee table. Due to lack of space, the engagement album goes to the storage, since it is the least recently referred to, i.e., LRU. With time, another child is born and the wedding album goes to storage and the second child's album takes its place. And so on so forth. At some later date, may be on your 25th wedding anniversary, you may look at your old wedding album and you get it from the storage. It stays in the living room till some other album takes its place. Whichever is the least recently referred to, will go to storage. This is exactly what happens in the *Db_Block* buffer. The data that is most recently used goes to the top while the least recently used goes to the bottom of the queue. If new data comes in, it enters on the top and the bottom most one, that is least recently used, is sent to the disk.

One may wonder why the bulletin board is required at all. All the information to be disseminated may be kept in a central (database) place for all employees to go and refer. The central place has all old as well as new bulletins (database). This will result in tremendous wastage of time if every employee has to go to the office, rummage through all the notices and get the most recent one. Instead it is lot faster if a copy of the only the most recent and important one is kept handy near a central place, i.e., the bulletin board. Similarly memory space access is much faster than disk access for a database. This is made use of in a sequence. In Oracle, one may recall that a *Sequence* is *created* by the following:

```
SQL> Create sequence abc
     minvalue 1 maxvalue 5000
     cache 10;
```

The *Sequence* is *created* in the database on the disk. Everytime you wish to draw a number from the *Sequence* "*abc*" you have to visit the database. Instead, while creating the *Sequence*, a Cache parameter 10 was chosen. That means a set of ten next following numbers are brought over from the database and kept handy in the memory space. Then the next ten numbers at any time are drawn from the memory space and this avoids as many trips (I/O traffic reduction) to the database. The memory space visits are much faster than the database visits.

1.2 Shared SQL Pool

In the Oracle 7 version, the above referred to *Data_Dictionary_Cache* is inside the *Shared_SQL_Pool*, which constitutes about 10-20% of the *SGA* size. In this pool, SQL statements run against the database are stored, in addition to their execution plan. When an identical SQL query is posed, this pool makes the execution of the query faster by providing the execution plan as well as the parse tree for the query.

1.3 Dictionary Cache

Data_Dictionary_Cache, as mentioned earlier, is an information link to the database. It has access to complete information about all database objects. If a *Table* is queried by a *User*, this cache is first referred to before the query is answered. The cache checks with the *Data_Dictionary Tables* in the database, if the database object of the query exists. If so, it checks where, and whether the *User* is authorised to query the object and whether the information asked for is in the object. Such information about that object is temporarily stored in the cache and such information grows. This information in the cache is also managed via LRU algorithm (explained above). Thus information about frequently referred to database objects exists in this cache for next *User* without repeating the visit to the database. This is how the process becomes faster. If the *Data_Dictionary_ Cache* is set too small (i.e., the *Data_Dictionary Table* information brought into it is very small), the *Data_Dictionary Tables* have to be queried often in the database. These queries are called **recursive hits**. Thus the *Data_Dictionary_Cache* is an important sentinel of the database for security. Recursive hits (requiring database visit) are obviously slower than the queries that are answered from the *Data_Dictionary_Cache* (memory visit).

1.4 Redolog Buffers

Oracle records all transactions made against the database to enable the database to be recovered completely in the event of an unexpected database crash. Ultimately these changes will go to the *Redolog* files from where they are stored for future reference. But much before they are written to the *Redolog* files, all transactions made to the database are temporarily stored (cached) in the *Redolog* buffers of the *SGA*. The information in the *Redolog* buffers is written periodically (rather than continuously) to the *Redolog* files.

1.5 SMON, PMON, DBWR, LGWR

The process areas of the memory structure of Oracle constitute mainly four processes. They are *SMON, PMON, DBWR and LGWR*. The job of the first memory process, *SMON*, a short form for server monitor, is to assist in *Instance* (Oracle organisation) recovery. When the database is started, *SMON* comes into action if there was an *Instance* crash earlier. It helps in the recovery of the

Instance in the event of a crash, for example, due to file-read or off-line errors. It recovers transactions skipped during crash. If any temporary segment is blocked at the time of crash and it is no longer required, it is released. Further, if there are contiguous (one by the side of the other) areas of free space, *SMON* coalesces them into one chunk. All transactional objects are eliminated. There is one and only one *SMON* for a given *Instance*.

The next memory process, *PMON*, a short form for process monitor, is responsible for clearing any failed transactions. For example, if an *update* fails before *commit* due to a crash, all cache areas are cleared up, *Table* locks are released and other process resources are freed. It rollsback uncommited transactions.

There is one and only one *PMON* per *Instance.* If either *SMON* or *PMON* dies, the *Instance* also dies.

The next memory process known as *DBWR***,** short form for dirty buffer writer, is important. It constantly watches two areas, the *Db_Buffer_*Cache and *Dictionary_Cache*. **The *DBWR*'s job is to write data from the buffer cache back to the database (disk) but it does not read data into the cache.** It is the job of the server process to read blocks from *Datafiles* and copy them into *SGA's Db_Buffer_Cache* and *Data_Dictionary_Cache*. Thus when a query is posed, the server process brings the data from the database into the *SGA*'s *Db_Buffer_Cache*. One copy of the data so retrieved from the database goes to *Rollback Segment*. The server process also writes in the *Data_Dictionary_Cache* the location of the records in the database from which the data was brought. This information is vital because after processing (like *update)* the data is to be returned to the original location from which it was brought. On updating the data, the changed data is first **copied** into *Redolog* buffers by the *DBWR*. The changed data in the *Db_Buffer_C*ache has to be returned to the database at the original location recorded in the *Data_Dictionary_Cache*. This transfer of changed data to database by the *DBWR* is not on a continuous basis. Instead, it is done periodically when the *Db_Block* buffer requires space to accommodate new incoming data. This implies that at a given time, the *Redolog* buffer contains a more up-to-date status of the database than the database itself.

This is simply because the *DBWR* may not have written the changed blocks back to the database since it was waiting for this writing. On *Commit*, the changed data in the *Redolog* buffer goes to the online *Redolog* files and the changed data in the *Db_Block* buffer has to go back to the database. If a *Rollback* command is given instead of *Commit*, the original data from the *Rollback Segment* is sent back to the database and the changed blocks in the *Redolog* buffer are purged out. The data retrieved from the database, irrespective of whether it was changed or not, has to return to the database.

If large queries are involved, multiple *DBWRs* will improve the performance. The *Db_Writer* parameter in the *Initn.Ora* file sets the number of *DBWR*s in the *Instance*. The job of *DBWR*, as the name suggests, is to write the changed blocks to the disk. The *DBWR* is responsible for providing sufficient space in the *Db_Block* buffer for new data.

If large amount of data is brought over into the *Db_Block* buffers for processing (*Update* etc.) and after processing, the data is kept uncommitted for too long, then it goes to *Rollback Segment* to be with the "before" image. This is done to make room in the *Db_Block* buffers for new data. So *Db_Block* buffer will not retain the data under process for too long. The *Rollback Segment* in this way operates like an extension of the *Db_Block* buffer.

The next memory process is *LGWR*, short form of log writer. As was mentioned earlier, in the *SGA*, the *Redolog* buffer cache accumulates the changed data. The job of *LGWR* is to write these changed data in batches from *Redolog* buffer cache to *Redolog* files. Thus the *Redolog* buffer cache entries are the most recent changed entries. While *DBWR*'s responsibility is to write changed data back to database files, *LGWR*'s is to write changed data cached in the *Redolog* buffer to *Redolog* files. This *LGWR* writing is done sequentially. It is important to note that *DBWR* accesses *Datafiles* in a random fashion. On *Commit*, although the changed blocks in *Db_Block* buffer are not written back to the disk, the changed data from *Redolog* buffers is written to *Redolog* files. *Commit* is considered complete, if and only if, the *Redolog* files are written to by the *LGWR*. *DBWR* writing into *Datafiles* is done only when more space is needed in the *Db_Block* buffers to receive new information.

In the *Db_Block* buffer, even if there is no change to the data, i.e., records are not changed but only rolled back, the data must be rewritten back to the disk. The *DBWR* writes to the database are deferred to optimise the I/O traffic.

There is an additional memory process known as *CKPT* (check point, which is like a wake up alarm) available in the later versions of Oracle. It periodically prompts the *DBWR* to write all changed data blocks to the *Datafiles* and also prompts the *LGWR* to transfer *Redolog* buffer entries into *Redologs*.

1.6 Redolog Files

Next on list is the *Redolog* files. They hold records on all the modifying transactions that occur in the database. That means the information required to recover a database after a crash is in the *Redolog* files. After *Update* or *Delete*, a *Commit* is not considered operative (*Complete* message given) unless all modifications are written to *Redolog* files. In such situations, locks are held for the duration of the transaction to prevent destructive interaction from other processes or transactions. A *Commit* or *Rollback* releases the lock on the data.

Every database will have two or more *Redolog* files. If multiple *Redolog* files are provided, the entries to these are written in cyclic fashion. That is, to start with, the first *Redolog* file is written to. When it fills up, the second one is written to and so on and so forth until the last one fills. Then the first one is again written to (overwritten). But before overwriting the first *Redolog* file, all the information in it is copied into a tape or disk tape. Only then the overwriting takes place. This type of copying of the *Redolog* files to tape is known as archiving. If for some reason archiving is not possible (such as, tape not available), and overwriting has to be done, then the database comes to a standstill until archiving is provided. That is how the database ensures that no information is lost. Thus *Redolog* files contain such important information, and to provide redundancy, each of the multiple *Redolog* files is always provided with a duplicate (mirrored). As a matter of abundant caution, each *Redolog* file in the pair is located on a different disk. Even if one is lost, the other survives to provide the information required for recovery after a crash. Although *Redolog* files are related to the database, they are stored outside the database so that in event of a database crash, they are not lost.

1.7 Control File

A database's overall physical architecture is maintained by its *Control* files. These record control information about all the files within the database. They are used to maintain internal consistency and guide recovery operation. Since these *Control* files are critical to the database, multiple copies of these are stored in the database. *Control* files are *created* at the time of creation of the database. It was mentioned earlier that *DBWR* writes all changed datablocks from the *Db_Block* buffers to the database at *CKPT*. At the same time, datafile *Headers* and *Control* files are also entered to record these changes to the datablocks. Thus *Control* files maintain transaction control (*SCN*: System change number) and datafile information. During recovery, *SCN* is required. *Control* file records the occurrence of changes to any *Datafiles* and the exact time when the change took place (time stamp). The *Control* file also has complete information about the physical location of all *Datafiles* in the database.

An Oracle *Instance* is a methodology by which any *User* (customer) accesses the Oracle database.

There are many similarities between an *Instance* and the way, for example, a restaurant operates. First we will look at the restaurant for simplicity.

1.8 Restaurant Example

Let us take the example of an Indian restaurant. We shall see how the restaurant is run. We will then illustrate how the situation in an

Oracle *Instance* is similar to that in a restaurant. In a restaurant, Kitchen is the database. Several *Tablespaces* exist in this kitchen database (structure) as shown in Illustration 1.3.

Tablespace	**Size**	**Data**	**No. of data**
Container 1	10 L	Bread Slices	500
Container 2	20 L	Potato Curry	15 Kg
Container 3	15 L	Egg Plant Curry	15 Kg
Container 4	20 L	Lentils	10 L
Container 5	20 L	Yogurt	10 L
Container 6	10 L	Rice	15 Kg
Container 7	20 L	Chicken Curry	15 Kg
Container 8	1 L	Coconut Chutney	1 L
Container 9	3 L	Soup	2 L
Container 10	5 L	Appetizer	3 Kg
Container 11	6 L	Tea	5 L
Container 12	6 L	Coffee	5 L

Illustration 1.3 - Restaurant Structure

Let us look at the dining table. To start with, assume that there are several tables and chairs. All are clean and vacant since the restaurant was just opened.

1.9 Standard_Meal.Ora

Assume one single client (standalone - one *User* per *Instance*) arrives and is led by the hostess to a two-seater table. He occupies the chair (Login). The waiter takes the order as *Standard_Meal*. The waiter assembles the dining ware by looking at the file *Standard_Meal.Ora* as shown in Illustration 1.4. In a *Standard_Meal*, you get 6 Bread slices, Potato Curry 100 g, Egg Plant Curry 100 g, Lentils 100 ml, Yogurt 100 ml and Rice 100 g.

The vessels are filled with appropriate items and served to the *User*. Since the Meal was ordered as per the average requirement of a single *User*, he consumes it all. At the signal from the *User*, the waiter brings the bill, collects the money and the *User* leaves (Logoff). The cleaner cleans the table and it is ready to receive another *User*. Here there is only one *User* at a time.

S No.	Parameter	No.	Required for
1	Num_Persons	1	
2.	Thali_Single	1	1
3.	Thali_Medium	0	2
4.	Thali_Large	0	6
5.	Large_Katori	0	
6.	Small_Katori	4	Four per person
7.	Spoon_Small	1	
8.	Water_Tumbler	1	
9.	Forks	0	
10.	Cup	0	
11	Serving Plate	0	

Illustration 1.4 - File Name: Standard_Meal.Ora for Standalone. (A Thali is a plate and a Katori is a bowl)

Instead of one, if you have a family of six people, the hostess will lead you to a six-seater table. (First step of the *Instance* determination). The waiter takes over from here. He solicits your order. Let us assume for your group of six, you order 33 Bread Slices, Potato Curry 500 g, Egg plant Curry 500 g, Lentils 500 ml, Yogurt 500 ml and Rice 500 g. Your family consists of a man, wife, grand-dad, grand-mom, boy and one girl. What you have ordered is a consolidated meal for all. Some may consume more of one item whereas others may consume more of a different item. It is meal for six based on average consumption. This type of ordering will cost much less than ala carte for each person. (consolidated requirement of six parallel *User*s in an *Instance*). So six people are going to share this *Instance* (*Shared Instance*).

1.10 Family_Meal.Ora

The waiter goes to the kitchen and assembles the dining ware for the *Family_Meal* as per the specifications in the file *Family_Meal.Ora* given in Illustration 1.5.

He fills the vessels with appropriate items ordered and serves them. Although the individual requirements may vary (e.g. man may eat more than the average number of bread slices, grand-dad and grand-mom may eat less), at the end, all the items get consumed. This is the principle of *Shared Instance*. The individual requirements of each *User* may slightly vary, but all the facilities offered get shared without much clash between the *Users*. Trouble comes only if any two members of the family (*Users*), demand a chunk much larger than the average of the same item in the *Family_Meal* and the quantity becomes insufficient. So in a *Shared Instance*, the requirements among individual *Users* should not be widely different for any resource.

S. No	Parameter No.	Required for	Remarks
1.	Num_Persons	6	
2.	Thali_Single	0	
3.	Thali_Medium	0	2
4.	Thali_Large	1	6
5.	Large_Katori	4	Curry1, Curry2, Lentils, Yogurt
6.	Small_Katori	24	Four per person
7.	Spoon_Small	6	
8.	Water_Tumbler	6	
9.	Forks	0	
10.	Cup	0	
11.	Serving Plate	6	

Illustration 1.5 - File Name: Family_Meal.Ora. (A Thali is a plate and a Katori is a bowl)

There is also another aspect to this *Shared Instance*. Since the order is for 33 bread slices, all are not served right in the beginning, since space in the plate is not sufficient to accommodate all bread slices. The waiter serves 10 bread slices first and replenishes as and when they get consumed. If for some reason the production of bread slices in the kitchen gets slowed down, the process of eating on the table also gets slowed down.

The satisfaction of the customers depends largely on the correctness of the order placed. If too large a number of bread slices is ordered, many will be left unconsumed. If too little a quantity of curry is ordered, there will be a shortage. The order should be commensurate with the average requirements of the members of the set of *Users*.

Now let us see what happens in Oracle. A single *User* logs in. He opens the *Init.Ora* file (similar to a menu card) and names it *Init1.Ora* (order copy) to identify his requirements out of the database. The *User* has to tell Oracle his requirement (corresponding to the order of the customer in the restaurant). In the restaurant example there are only a few parameters. Oracle, which is capable of doing many tasks, in its later version has nearly 190 parameters. Fortunately, for an average *User*, most of the parameters are left at their default values and only a few parameters need to be specified. Even in a typical restaurant, there are default parameters. For example, water need not be ordered. It comes by default. If wrong parameters are chosen (unbalanced order in the restaurant), the performance of Oracle will be poor and *User* will not be satisfied with it. The choice of parameters is dictated by the way Oracle is used by the *User*. For example, if the *User* application is purely a decision support (*DSS*), i.e., it mostly is used to extract large information from the database by a single *User*, a large *RBS* is required assuming that the `Select` statement is used for a large number of records at a time. On the other hand, if the application needs frequent *updates (OLTP)*, smaller *Rollback Segment* is to be chosen in the *Init1.Ora*. This is because, *RBS* has to keep a copy of the set of records before *Update* till the data is *committed* or *rolled back*. Suppose, your *System* does not have a provision for an appropriate *RBS* for *OLTP*, then it will get overloaded. Similarly, if the application involves a large number of *Deletes/Updates*, then more *Redolog* files and *Archivelogs* have to be provided.

Thus, it is clear that allocation of different memory spaces in the *SGA* is very critical and the actual allocation depends on each application. Hence different types of *Instances* are to be configured for different types of applications. Otherwise a via-media approach of *Instance* may be designed to suit the requirements of a set of parallel *Users*.

1.11 Instance

In Oracle, using *Instance*, you can provide a plan of the configuration of several parameters for the shape of different memory structures etc. through which the database is accessed. Apriori, in order to efficiently get the job done through an *Instance*, you should first know what you want to do with the database after you access it. **There is no universal *Instance* which does all jobs with equal efficiency**. What all this means is, to have an equally efficient *Instance* for all jobs, you need an infinitely large memory. This can never be the case. The trick lies in managing the *Instance* to get the best results from the database with limited resources. This is like driving a car up or down a hill. The *Instance* used to go up the hill, (i.e., parameters are: low gear and foot on gas pedal) is different from the *Instance* used for going down the hill (i.e., slightly higher gear with foot on the brake). If you use the wrong parameters, the results of driving are disastrous. If you drive down hill in low gear, with foot on gas pedal, the car will crash. (*Instance* will crash).

So before you access a database, you must tell the *DBA* what you wish to do with the database. He should also be aware of the details of your application program so that he can assist you in getting better results. He can suggest minor changes in your software to manage with the limited resources set apart in that *Instance* in which your application fits best. Suppose you wish to *update* a large number of records, you may call all the records in one go, apply the change, and then *commit*. This requires an enoromous *RBS* space. Instead, you may call the records say 20 at a time, apply change and *commit* them. The requirement of *RBS* space in this case is considerably smaller. As was mentioned earlier, if you have an infinitely large memory area (which is not