dbazine.com

bmcsoftware

Oracle Space Management Handbook

with a foreword by Don Burleson

Donald K. Burleson
Dave Ensor
Christopher Foot
Lisa Hernandez
Mike Hordila
Jonathan Lewis
Dave Moore
Arup Nanda
John Weeg

ISBN: 0-9744355-0-3
Retail Price $24.95 US/$37.95 Canada

Oracle Space Management Handbook

Donald K. Burleson
Dave Ensor
Christopher Foot
Lisa Hernandez
Mike Hordila
Jonathan Lewis
Dave Moore
Arup Nanda
John Weeg

RAMPANT TECHPRESS

Oracle Space Management Handbook

By: Donald K. Burleson, Dave Ensor, Christopher Foot, Lisa Hernandez, Mike Hordila, Jonathan Lewis, Dave Moore, Arup Nanda, John Weeg

Printed in the United States of America.

Series Editor: Donald K. Burleson

Production Manager: John Lavender

Production Editor: Teri Wade

Cover Design: Bryan Hoff

Printing History:

August, 2003 for First Edition

ISBN: 0-9744355-0-3

Table of Contents

Conventions Used in this Book

It is critical for any technical publication to follow rigorous standards and employ consistent punctuation conventions to make the text easy to read.

However, this is not an easy task. Within Oracle there are many types of notation that can confuse a reader. Some Oracle utilities such as STATSPACK and TKPROF are always spelled in CAPITAL letters, while Oracle parameters and procedures have varying naming conventions in the Oracle documentation. It is also important to remember that many Oracle commands are case sensitive, and are always left in their original executable form, and never altered with italics or capitalization.

Hence, all Rampant TechPress books follow these conventions:

Parameters - All Oracle parameters will be *lowercase italics*. Exceptions to this rule are parameter arguments that are commonly capitalized (KEEP pool, TKPROF), these will be left in ALL CAPS.

Variables – All PL/SQL program variables and arguments will also remain in lowercase italics (*dbms_job, dbms_utility*).

Tables & dictionary objects – All data dictionary objects are referenced in lowercase italics (*dba_indexes, v$sql*). This includes all v$ and x$ views (*x$kcbcbh, v$parameter*) and dictionary views (*dba_tables, user_indexes*).

SQL – All SQL is formatted for easy use in the code depot, and all SQL is displayed in lowercase. The main SQL terms (select, from, where, group by, order by, having) will always appear on a separate line.

Programs & Products – All products and programs that are known to the author are capitalized according to the vendor specifications (IBM, DBXray, etc). All names known by Rampant TechPress to be trademark names appear in this text as initial caps. References to UNIX are always made in uppercase.

About the Authors

Donald K. Burleson is one of the world's top Oracle Database experts with more than 20 years of full-time DBA experience. He specializes in creating database architectures for very large online databases and he has worked with some of the world's most powerful and complex systems. A former Adjunct Professor, Don Burleson has written 15 books, published more than 100 articles in national magazines, serves as Editor-in-Chief of Oracle Internals and edits for Rampant TechPress. Don is a popular lecturer and teacher and is a frequent speaker at Oracle Openworld and other international database conferences.

Christopher T. Foot is an Oracle certified senior-level instructor, technical sales specialist and database architect for Contemporary Technologies Inc. He has fifteen years' experience with database Technologies and is a regular speaker at the International Oracle Users Group and Oracle Open World conferences Contemporary Technologies Inc. is a leading provider of Oracle products and services.

Dave Ensor is a Product Developer with BMC Software where his mission is to produce software solutions that automate Oracle performance tuning. He has been tuning Oracle for 13 years, and in total he has over 30 years active programming and design experience.

As an Oracle design and tuning specialist Dave built a global reputation both for finding cost-effective solutions to Oracle performance problems and for his ability to explain performance issues to technical audiences. He is co-author of the O'Reilly & Associates books Oracle Design and Oracle8 Design Tips.

Liza Fernandez is an aspiring DBA working toward her Oracle 9i DBA certification. She is also pursuing her Master's Degree in Information Systems Management.

Mike Hordila is a DBA OCP v.7, 8, 8i, 9i, and has his own Oracle consulting company, DBActions Inc., www.dbactions.com, in Toronto, Ontario. He specializes in tuning, automation, security, and very large databases. Mike has articles in Oracle Magazine Online, Oracle Internals and DBAzine.com. Updated versions of his work are available on www.hordila.com. He is also a technical editor with Hungry Minds (formerly IDG Books).

Jonathan Lewis is a freelance consultant with more than 17 years experience in Oracle. He specializes in physical database design and the strategic use of the Oracle database engine, is author of *Practical Oracle 8i - Building Efficient Databases* published by Addison-Wesley, and is one of the best-known speakers on the UK Oracle circuit. Further details of his published papers, tutorials, and seminars can be found at www.jlcomp.demon.co.uk, which also hosts *The Co-operative Oracle Users' FAQ* for the Oracle-related Usenet newsgroups.

Dave Moore is a product architect at BMC Software in Austin, TX. He's also a Java and PL/SQL developer and Oracle DBA.

Arup Nanda is the founder and President of Proligence, a specialized Oracle database services provider in the New York metropolitan area, which provides tactical solutions in all aspects of the an Oracle project life cycle. He has been an Oracle DBA for more than nine years has touched almost all types of database performance issues. He specializes in Oracle performance evaluation and high availability solutions.

John Weeg has over 20 years of experience in information technology, starting as an application developer and progressing to his current level as an expert Oracle DBA. His focus for the past three years has been on performance, reliability, stability, and high availability of Oracle databases. Prior to this, he spent four years designing and creating data warehouses in Oracle. John can be reached at jweeg@hesaonline.com or http://www.hesaonline.com/dba/dba_services.shtml.

Foreword

As a database management system, the management of Oracle file structures is critically important to the successful operation of any system. The Oracle administrator must understand all levels of Oracle file management, including data file management principles, tablespace management principles, and the storage of individual objects within the tablespaces. As Oracle has evolved into one of the world's most complex database management systems, it is imperative that all Oracle professionals understand how their information is stored both at the logical and physical level. The purpose of this book is to bring together some of the world's best experts to talk about storage management internals and to provide you with insights into the successful operation of large complex Oracle databases.

This book is designed to provide you with specific insights and techniques that you can use to immediately be successful within your Oracle enterprise. Given the amazing wealth of choices that Oracle offers with regard to data storage mechanisms, it is important for the Oracle professional to understand which mechanisms are appropriate, and not appropriate, for their specific database needs. The focus of this book is going to be about how you can leverage Oracle's wealth of choices in order to choose the optimal configuration for your I/O subsystem.

This book will review Space Management in six functional areas: Data Files, Tablespaces, Tables, Indexes, Partitioning, and Replication.

In the section on space management as it applies to data files, this text covers relevant topics such as I/O contention, determining which files need resizing and the optimum size to make them, tuning to reduce disk I/O, using *v$segstat* and *v$segment_statistics* to isolate segment level problems, index compression and Index Organized Tables (IOT), simplifying the process of verifying that your backup ran successfully, Interested Transaction List (ITL) waits, and what to consider before re-writing SQL statements to try to save CPU costs.

Space management for tablespaces offers a PL/SQL package to automate database cleanup, a thorough discussion of TEMP tablespaces, a section on the ability of Oracle's dataserver to manage itself, strategies for using locally-managed tablespaces, and a discussion of Oracle's ability to support multiple block sizes

In the discussion on space management for tables you will read about automating periodic table and index reorganization, and the practical application, management, and performance issues of external tables.

This text also covers how to rebuild indexes without worrying about fragmentation, and how to size indexes for new and existing tables. There is a discussion on how to partition tables and then some of the perils and pitfalls to watch for. The text wraps up with a discussion on automating replication.

We hope you will be able to apply some of the techniques offered in this text to your production environment to enhance the success of your Oracle enterprise.

CHAPTER 1

Measuring Oracle Segment I/O

What is Really Going On?

We are taught, from the beginning, that we want to separate our tablespaces onto different mount points or drives to minimize I/O level contention. There are devices that minimize this physical level contention; but most of my smaller clients don't have these devices, so we still have this concern. How do we see what is really at the file level? Given a smaller system, where I can fit the majority of the database into memory; does the I/O matter?

Theory

Ideally all the data we want to work with are in memory and no I/O is needed. In reality, you usually can't count on this being the case. So, our goal is to try to minimize the disk physical movement for any given data request. For example, if the index and the data are on the same disk, there is movement needed for the index and then the same disk must move for the data. If the next read wants the next record, then we must move back for the index and back again for the data. We have made the read for the data and the read for the index get in each other's way.

The theory says that all objects that might be used in the same transaction should be on different mount points. So we have the following minimum mount points:

- System tablespace
- Data tablespace
- Index tablespace
- Rollback segments
- Archive logs
- Temporary tablespace

These six mount points would give us our basic good system setup. Let's put this theory to the test.

Test It

There are two very useful dynamic system views, *v$datafile* and *v$tempfile*, that will allow us to test this theory. Previous to 8.1 you won't find *v$tempfile*. These both have the same layout, so I will just work with *v$datafile* for the testing. The fields we are interested in first are the number of physical reads/writes and the number of blocks read and written.

This view gives the statistics since instance startup so we have created a table to isolate and compare the results of *v$datafile* for the current activity:

```
create table phy_io tablespace data01
storage (initial 64k next 64k pctincrease 0) pctfree 0 pctused 80
unrecoverable
as select file#,phyrds,phywrts,phyblkrd,phyblkwrt from v$filestat;
```

So let's see what we have right now in a newly started instance - the only activity has been to create this table:

```
SQL> select c.file_name,a.file#,a.phyrds-b.phyrds phyrds,a.phywrts-
b.phywrts phywrts
FILE_NAME (IO)        FILE#     PHYRDS    PHYWRTS   PHYBLKRD  PHYBLKWRT
---------------- ---------- ---------- ---------- ---------- ----------
SYSTEM01.DBF              1         29         26         47         26
DATA0101.DBF              3          1          1          1          1
```

The I/O against DATA0101.DBF is me accessing the *phy_io* table. If we check memory we can see the current blocks:

```
SQL> select b.file_name,a.file#,a.cnt from
  2  (select file#,count(1) cnt from v$bh group by file#) a
  3  ,dba_data_files b
  4  where a.file#=b.file_id;

FILE_NAME (IN MEMORY)       FILE#        CNT
--------------------- ---------- ----------
SYSTEM01.DBF                   1        569
RBS01.DBF                      2         11
DATA0101.DBF                   3          2
```

Okay, so we see how the system starts. Now, if we access some data, what happens?

```
SQL> select dsn,subst_id from iuc4.ds_admin_tab where dsn=523;

       DSN SUBST_ID
---------- -----------
       523 101316-69-2
```

Checking our I/O now we see there were four physical reads to get this data in and we have four additional blocks allocated in memory:

```
FILE_NAME (IO)        FILE#     PHYRDS    PHYWRTS   PHYBLKRD  PHYBLKWRT
---------------- ---------- ---------- ---------- ---------- ----------
SYSTEM01.DBF              1         59         52         92         52
DATA0101.DBF              3          5          1          5          1

FILE_NAME (IN MEMORY)       FILE#        CNT
--------------------- ---------- ----------
SYSTEM01.DBF                   1        587
RBS01.DBF                      2         11
DATA0101.DBF                   3          6
```

These four new blocks in memory are the data and index:

```
SQL> select b.owner,b.object_name,a.cnt from
  2  (select objd,count(1) cnt from v$bh group by objd) a
  3  ,dba_objects b
  4  where b.data_object_id = a.objd and b.owner = 'IUC4';

OWNER           OBJECT_NAME                      CNT
--------------- ------------------------- ----------
IUC4            DS_ADMIN_TAB                       2
IUC4            DS_ADMIN_IDX1                      2
```

To bring this data in, we performed four separate read actions and we see that we needed to take two separate objects, table and index, from the same file so we were contending with ourselves. We also see that there was additional I/O against the system tablespace to get the object definition. In addition, there is I/O going on against the system tablespace for timing, so you will see that number rise even when there is no activity.

What Happens When We Update?

```
SQL> update iuc4.ds_admin_tab set subst_id = '101316-69-2' where
dsn=523;
1 row updated.
```

I also committed!

```
FILE_NAME (IO)        FILE#     PHYRDS    PHYWRTS   PHYBLKRD  PHYBLKWRT
---------------- ---------- ---------- ---------- ---------- ----------
SYSTEM01.DBF              1         70         78        118         78
DATA0101.DBF              3          5          1          5          1

FILE_NAME (IN MEMORY)       FILE#        CNT
---------------------- ---------- ----------
SYSTEM01.DBF                    1        591
RBS01.DBF                       2         11
DATA0101.DBF                    3          6
```

Nothing happened! We see there is no physical change. I expected I/O at least to the rollback file. If I rollback instead of commit, there is still no I/O count change. So we are seeing that the system really wants to work just in memory whenever it can.

Now let's force the issue with a checkpoint:

```
SQL> alter system checkpoint;
System altered.
FILE_NAME (IO)        FILE#     PHYRDS    PHYWRTS   PHYBLKRD  PHYBLKWRT
---------------- ---------- ---------- ---------- ---------- ----------
SYSTEM01.DBF              1         71        103        119        103
RBS01.DBF                 2          1         12          1         12
DATA0101.DBF              3          6          4          6          4

FILE_NAME(IN MEMORY)       FILE#        CNT
--------------------- ---------- ----------
SYSTEM01.DBF                   1        591
RBS01.DBF                      2         11
DATA0101.DBF                   3          6
```

Here we see the write contention. We performed a write against all three files. If they are on the same mount point, then it happened serially --they wrote single threaded.

What Else?

The *v$filestat* also will tell us the time spent performing reads and writes, in hundredths of a second, if *timed_statistics* is set to true. When I look at a system that has been up for a longer time, I see that the average time to write a block of data is about ten times longer than the average time to read a block.

So What?

Take a look at your *v$filestat* and *v$tempstat* views. Mine have shown me that even though Oracle works in memory as much as possible, I still need to be very aware of I/O-level contention. I also see that wherever I can, I will try to minimize the number of write actions performed.

Watch yours for a while to see what is really going on.

Datafile Resizing Tips

CHAPTER 2

Setting Free Your Space

Conventional database wisdom dictates that we should treat disk space as though it were free, unfettered space waiting to be filled with data. It's a liberating idea, but not entirely practical. Not surprisingly, it's also rife with potential problems. Get too comfortable with the idea of space being "free" and you may suddenly find yourself scrounging around for it.

When setting up a database, for example, a database administrator usually takes educated guesses at the company's space needs, large or small. It's important to point out, however, that those guesses tend to be conservative. The reason? By overestimating the amount of space needed, the administrator is less likely to wind up stuck when he or she first loads all the data. Once the instance runs for a while, it's possible to see just how far off the original estimate of space requirements was. Moreover, the administrator can give back some of that space.

Over-allocation of space at the file level affects the backup/recovery window, file checking times and, most painfully, limits the potential allocation of space to a tablespace that needs the extra room. A simpler solution would be to review the evolution of the script, which lets the administrator know which files can and cannot be resized to create more space.

Alter Database

It's possible to release space from data files but only down to the first block of data. This is done with the 'alter database' command. Rather than go through the tedious process of manually figuring out the command every time it's used, it makes more sense to write a script that will generate this command as needed.

The basic syntax for this command is:

```
Alter database name datafile 'file_name' resize size;
```

Where *name* is the name of the database, *file_name* is the name of the file and *size* is the new size to make this file. We can see this size change in the *dba_data_files* table as well as from the server.

First, pull in the database name:

```
Select 'alter database '||a.name
From v$database a;
```

Once that has been done, it's time to add in data files:

```
select 'alter database '||a.name||' datafile '''||b.file_name||''''
from v$database a
,dba_data_files b;
```

While this is closer to the ultimate solution, it's not quite there yet. The question remains: Which data files do you want to alter? At this point, you can use a generally accepted standard, which allows tablespaces to be 70 percent to 90 percent full. If a tablespace is below the 70 percent mark, one way to bring the number up is to de-allocate some of the space.

So how do you achieve percent full? While there are a number of different ways, simple is usually ideal. Here's how it works.

Amount used:

```
Select tablespace_name,sum(bytes) bytes_full
From dba_extents
Group by tablespace_name;
```

Total available:

```
Select tablespace_name,sum(bytes) bytes_total
From dba_data_files
Group by tablespace_name;
```

So if we add this with our original statement, we can select on *pct_used* (less than 70 percent):

```
select 'alter database '||a.name||' datafile '''||b.file_name||''''
from v$database a
,dba_data_files b
,(Select tablespace_name,sum(bytes) bytes_full
From dba_extents
Group by tablespace_name) c
,(Select tablespace_name,sum(bytes) bytes_total
From dba_data_files
Group by tablespace_name) d
Where b.tablespace_name = c.tablespace_name
And b.tablespace_name = d.tablespace_name
And bytes_full/bytes_total < .7
;
```

According to the command, a selection has been made based on tablespace. What if you want to resize based on file? It's crucial to remember that multiple files can exist in any tablespace. Plus, only space that is after the last data block can be de-allocated. So the next step should be to find the last data block:

```
select tablespace_name,file_id,max(block_id) max_data_block_id
from dba_extents
group by tablespace_name,file_id;
```

Now that the command to find the last data block has been inserted, it is time to find the free space in each file above that last data block:

```
Select a.tablespace_name,a.file_id,b.bytes bytes_free
From (select tablespace_name,file_id,max(block_id) max_data_block_id
from dba_extents
group by tablespace_name,file_id) a
,dba_free_space b
where a.tablespace_name = b.tablespace_name
and a.file_id = b.file_id
and b.block_id > a.max_data_block_id;
```

So far, so good. How is it possible, then, to combine commands to ensure the correct amount will be resized? In fact, it's fairly easy.

```
select 'alter database '||a.name||' datafile '''||b.file_name||'''' ||
' resize '||(bytes_total-bytes_free)
from v$database a
,dba_data_files b
,(Select tablespace_name,sum(bytes) bytes_full
From dba_extents
Group by tablespace_name) c
,(Select tablespace_name,sum(bytes) bytes_total
From dba_data_files
Group by tablespace_name) d
,(Select a.tablespace_name,a.file_id,b.bytes bytes_free
From (select tablespace_name,file_id
,max(block_id) max_data_block_id
from dba_extents
group by tablespace_name,file_id) a
,dba_free_space b
where a.tablespace_name = b.tablespace_name
and a.file_id = b.file_id
and b.block_id > a.max_data_block_id) e
Where b.tablespace_name = c.tablespace_name
And b.tablespace_name = d.tablespace_name
```

Double Checking the Work

Now, the thing to do is ensure that the right amount of space - not too much, not too little - has been de-allocated. The rule of thumb to follow: Do not go above 70 percent of the tablespace being used. If you have already pulled out how much is used from *dba_extents*, you can simply add a check to your statement:

```
select 'alter database '||a.name||' datafile '''||b.file_name||'''' ||
' resize '||greatest(trunc(bytes_full/.7)
,(bytes_total-bytes_free))
from v$database a
,dba_data_files b
,(Select tablespace_name,sum(bytes) bytes_full
From dba_extents
Group by tablespace_name) c
,(Select tablespace_name,sum(bytes) bytes_total
From dba_data_files
Group by tablespace_name) d
,(Select a.tablespace_name,a.file_id,b.bytes bytes_free
From (select tablespace_name,file_id
,max(block_id) max_data_block_id
from dba_extents
group by tablespace_name,file_id) a
,dba_free_space b
where a.tablespace_name = b.tablespace_name
and a.file_id = b.file_id
and b.block_id > a.max_data_block_id) e
Where b.tablespace_name = c.tablespace_name
And b.tablespace_name = d.tablespace_name
And bytes_full/bytes_total < .7
And b.tablespace_name = e.tablespace_name
And b.file_id = e.file_id
;
```

One last thing to do: Add a statement to indicate what is being changed.

```
select 'alter database '||a.name||' datafile '''||b.file_name||'''' ||
' resize '||greatest(trunc(bytes_full/.7)
,(bytes_total-bytes_free))||chr(10)||
'--tablespace was '||trunc(bytes_full*100/bytes_total)||
'% full now '||
trunc(bytes_full*100/greatest(trunc(bytes_full/.7)
,(bytes_total-bytes_free)))||'%'
from v$database a
,dba_data_files b
,(Select tablespace_name,sum(bytes) bytes_full
From dba_extents
Group by tablespace_name) c
,(Select tablespace_name,sum(bytes) bytes_total
From dba_data_files
Group by tablespace_name) d
,(Select a.tablespace_name,a.file_id,b.bytes bytes_free
From (select tablespace_name,file_id
,max(block_id) max_data_block_id
from dba_extents
group by tablespace_name,file_id) a
,dba_free_space b
where a.tablespace_name = b.tablespace_name
and a.file_id = b.file_id
and b.block_id > a.max_data_block_id) e
Where b.tablespace_name = c.tablespace_name
And b.tablespace_name = d.tablespace_name
And bytes_full/bytes_total < .7
And b.tablespace_name = e.tablespace_name
And b.file_id = e.file_id
;
```

At last, here's a script that will create the script. Even so, it's important to pay careful attention when applying the created script. Why? Because Rollback, System and Temporary tablespaces are vastly different creatures, and each should not necessarily be held to the 70 percent rule. By the same token, there might be a very good reason for a tablespace to be over allocated -- like the giant load that will triple the volume tonight.

A word of caution, too: Be sure that extents can still be allocated in each tablespace. There may be enough free space, but it may be too fragmented to be useful. That problem will be the focus of another article.

Reducing Disk I/O on Oracle Datafiles

CHAPTER 3

Oracle Expert Tuning Secrets to reduce disk I/O

In this installment, we will examine disk I/O and understand how reducing disk I/O s the single most important Oracle tuning activity.

Oracle tuning and Disk I/O

Disk I/O is a very time-consuming task, and almost every Oracle tuning activity has the ultimate goal of reducing disk I/O.

When we look at Oracle9i tuning, we see that almost every tuning activity is done with the ultimate goal of reducing disk I/O. To make this clear, let's look at some common tuning activities and see how they reduce disk I/O:

Tuning SQL statements - When we tune an SQL statement to replace a full-table scan with an index range scan, the performance improvement is the direct result of a reduction in disk I/O.

Changes to the Oracle SGA - When we increase the *shared_pool*, *large_pool*, or *db_cache_size*, the resulting performance improvement is related to the reduction in disk I/O.

Table reorganizations - When we reorganize a table, we remove extent fragments, coalesce chained rows, re-build the freelist chain, and re-sequence table rows. These activities all have the result of reducing the amount of disk I/O in the Oracle database.

In sum, Disk I/O is the single most expensive operation within an Oracle9i database, and multiple block sizes give us a powerful new tool to manage disk I/O with more power than ever before.

Let's see how using Oracle RAM data buffers help to reduce disk I/O.

Tuning with RAM Data Buffers

In Oracle9i we have the ability to define up to seven separate and distinct data buffers. These data buffers can be used to isolate Oracle data in RAM and improve performance by reducing disk I/O.

These buffers can have different block sizes and named pools exist for 2K, 4K, 16K and 32K buffers and we also have three other pools, the default pool, the recycle pool and the keep pool.

Let's take a look at each of these data buffers.

The KEEP Pool

When the KEEP was first introduced in Oracle8i, its purpose was to provide a RAM data buffer to fully-cache blocks frequently referenced tables and indexes. For example, when computing the size of the KEEP pool, we must total the number of bytes for all tables that have been marked to reside in the KEEP pool. This is because we always want the KEEP pool to fully cache all tables that have been assigned to the KEEP pool.

In Oracle9i, a table must reside in a tablespace of the same block size as the cache assigned to the table.

```
alter table CUSTOMER storage (buffer_pool KEEP);
```

Remember, the point of the KEEP pool is to always have a data buffer hit ratio of 100%. Also note that the block size of the KEEP pool is not important. This is because, once loaded, all blocks in the KEEP pool will always remain in RAM memory. In our example, the KEEP poll is a 32K blocksize because we wanted the RECYCLE pool to have a large block size to improve the performance of full-table scans.

Locating Tables and Indexes for the KEEP Pool

The Oracle documentation states "A good candidate for a segment to put into the KEEP pool is a segment that is smaller than 10% of the size of the DEFAULT buffer pool and has incurred at least 1% of the total I/Os in the system." In other words, small, highly accessed tables are good candidates for caching.

So, how do we identify small-table full table scans? The best method is to explain all of the SQL that is currently in your library cache and then generate a report showing all of the full table scans in your database at that time. I invented a very important script called *access.sql* that was published in the December 2000 issues of Oracle magazine. Here is the link:

```
http://www.oracle.com/oramag/oracle/00-nov/index.html?o60dba.html
```

Running the *access.sql* script should give us all of the information we need to identify tables for the KEEP pool. Any small tables (for example, less than 50 blocks) that have a high number of full table scans will benefit from being added to the KEEP pool. In the report below, we see output from an Oracle Applications database, and we see full table scans on both large and small tables.

```
 full table          scans and counts

full table scans and counts

OWNER      NAME                         NUM_ROWS C K   BLOCKS  NBR_FTS
---------- ------------------------ ------------ - - -------- --------
APPLSYS    FND_CONC_RELEASE_DISJS             39 N K        2   98,864
APPLSYS    FND_CONC_RELEASE_PERIODS           39 N K        2   98,864
APPLSYS    FND_CONC_RELEASE_STATES             1 N K        2   98,864
SYS        DUAL                                  N K        2   63,466
APPLSYS    FND_CONC_PP_ACTIONS             7,021 N      1,262   52,036
APPLSYS    FND_CONC_REL_CONJ_MEMBER            0 N K       22   50,174
APPLSYS    FND_CONC_REL_DISJ_MEMBER           39 N K        2   50,174
APPLSYS    FND_FILE_TEMP                       0 N         22   48,611
APPLSYS    FND_RUN_REQUESTS                   99 N         32   48,606
INV        MTL_PARAMETERS                      6 N K        6   21,478
APPLSYS    FND_PRODUCT_GROUPS                  1 N          2   12,555
APPLSYS    FND_CONCURRENT_QUEUES_TL           13 N K       10   12,257
AP         AP_SYSTEM_PARAMETERS_ALL            1 N K        6    4,521
APPLSYS    FND_CONCURRENT_QUEUES              13 N K       10    4,078
```

From examining the above report, we identify the following files for addition to the KEEP pool. We select those tables with less than 50 blocks that are not already in the KEEP pool (the "K" column).

```
OWNER          NAME                              NUM_ROWS C K   BLOCKS
NBR_FTS
-------------- ------------------------ ---- -------- - - -------- ----
----
PPLSYS         FND_FILE_TEMP                           10 N          22
48,611
APPLSYS        FND_RUN_REQUESTS                        99 N          32
48,606
APPLSYS        FND_PRODUCT_GROUPS                       1 N           2
12,555
```

Remember, our goal is for the data buffer hit ratio for the KEEP pool to always be 100 percent. Every time we add a table to the KEEP pool, we must also add the number of blocks in the table to the KEEP pool parameter in our *init.ora* file.

Once you have explained all of the SQL in your library cache, you will have a plan table with all of the execution plans and an *sqltemp* table with all of the SQL source code. Once these tables are populated, you can run a script to generate the KEEP syntax for you. Let's take a look at this script:

The RECYCLE Pool

This data pool is reserved for large-table full table scans. Because Oracle data blocks from full table scans are unlikely to be reread, the RECYCLE pool is used so that the incoming data blocks do not "flush out" data blocks from more frequently used tables and indexes. Large tables that experience full-table scans are assigned to the RECYCLE pool to prevent their data blocks from reducing available blocks for other tables.

Now let's see how multiple block sizes can improve Oracle performance.

Using Multiple Block Sizes

The most important consideration when using multiple block sizes in Oracle9i is to segregate different portions of the Oracle database into different data pools.

When an SQL statement requests the fetch of a result set from Oracle tables, the SQL is probably retrieving the table by an index.

As an Oracle8i tuning expert, I often recommended that a whole database be re-defined with a large blocksize. Many people were mystified when a database with a 2K-block size was increased to an 8K-block size and the entire database ran faster. A common justification for resisting a block size increase was "This database randomly fetches small rows. I can't see why moving to a larger block size would improve performance." So, then, what explains the performance improvement with larger block sizes?

When choosing a block size, many DBAs forget about the index trees and how Oracle indexes are accessed sequentially when doing an index range scan. An index range scan is commonly seen in nested loop joins, and the vast majority of row access involved indexes.

Because index range scans involve gathering sequential index nodes, placing the indexes in a larger block size reduces disk I/O and improves throughput for the whole database.

So then, why not create our entire Oracle database with large block sizes and forget about multiple block sizes? The answer is not simple. In order to fully utilize the RAM memory in the data buffers, you must segregate tables according to their distribution of related data.

Small blocks - Tables with small rows that are accessed in a random fashion should be placed onto tablespaces with small block sizes. With random access and small block sizes, more of the RAM in the data buffer remains available to hold frequently referenced rows from other tables.

Large blocks - Row-ordered tables, single-table clusters, and table with frequent full-table scans should reside in tablespaces with large block sizes. This is because a single I/O will fetch many related rows and subsequent requests for the "next" rows will already be in the data buffer.

The goal here is simple; we want to maximize the amount of available RAM memory for the data buffers by setting the block sizes according to the amount of I/O experienced by the table or index. Random access of small rows suggests small block sizes, while sequential access of related rows suggests large block sizes.

Here is a sample of an Oracle *init.ora* file that uses separate data buffers with different block sizes:

```
db_block_size=32768          -- This is the system-wide
                             -- default block size

db_cache_size=3G             -- This allocates a total of 3 gigabytes
                             -- for all of the 32K data buffers

db_keep_cache_size=1G        -- Here we use 1 gigabyte for the KEEP pool

db_recycle_cache_size=500M   -- Here is 500 meg for the RECYCLE pool
                             -- Hence, the DEFAULT pool is 1,500 meg

-- *****************************************************************
-- The caches below are all additional RAM memory (total=3.1 gig)
-- that are above and beyond the allocation from db_cache_size
-- *****************************************************************

db_2k_cache_size=200M        -- This cache is reserved for random
                             -- block retrieval on tables that
                             -- have small rows.

db_4k_cache_size=500M        -- This 4K buffer will be reserved
                             -- exclusively for the data dictionary.
                             -- Only the SYSTEM tablespace has 4K blocks

db_8k_cache_size=800M        -- This is a separate cache for
                             -- segregating I/O for specific tables

db_16k_cache_size=1600M      -- This is a separate cache for
                             -- segregating I/O for specific tables
```

Next let's move deeper and explore techniques for identifying hot data files within Oracle. By knowing those data files that have lot's of I/O, we gain insight toward our goal of reducing I/O.

Disk I/O Tuning

For other operating environments, we are concerned whenever we see a backlog of I/O tasks waiting to access data on a single disk. For other operating systems, the *iostat* utility can be used to detect I/O issues.

Once you've identified the hot disks, look closely to find out which files and tables on the disks experience most of the activity, so that you can move them to less-active disks as needed. The actual process of identifying hot files and disks involves running data collection utilities, such as STATSPACK and the UNIX *iostat* utility, and then using the collected I/O data to pinpoint the sources of excessive I/O measurements.

Here are the cardinal rules for disk I/O:

Understand I/O - There is a difference between a busy disk and a disk that is waiting for I/O to complete. In the next section we will explore the UNIX *iostat* utility and show how you can identify busy disks.

Monitor disk I/O - Many disk arrays such as EMC provide sophisticated disk monitoring tools such as Open Symmetrics Manager and Navistar. These tools report on more than simple disk waits, and highlight contention for disks, channels, and disk adapters.

Use RAID properly - If you are using RAID such as RAID 0+1, the Oracle data blocks will be spread randomly across all of the disks, and load will rise and fall in a uniform fashion.

Control where disk I/O happens - Senior Oracle DBAs often prefer not to implement RAID striping so that they have more control over the disk I/O subsystem.

Now that we understand the basic principles behind locating hot disks, let's see how STATSPACK can be extended to capture file I/O information.

STATSPACK Reports for Oracle Datafiles

To perform I/O load balancing, we need to get information about the amount of I/O for an Oracle datafile, relative to the total I/O from the database. Remember, a hot file is not necessarily causing a disk bottleneck. The goal of the STATSPACK technique below is to alert the Oracle DBA to those datafiles that are taking a disproportionate amount of I/O relative to other files in the database.

The script we use for this purpose is called *rpt_hot_files.sql*, and this script is also incorporated into our generalized DBA alert script, *statspack_alert.sql.*

The *rpt_hot_files.sql* script is listed below.

To get the data we need, we rely on two STATSPACK tables:

stats$sysstat The *stats$sysstat* table contains two important metrics. These are used to compute the total read I/O and write I/O for the entire database:

- Total physical reads (statistic#=40)
- Total physical writes (statistic#=44)

stats$filestatxs The *stats$filestatxs* table contains detailed read I/O and write I/O, totaled by datafile name.

We then compare the system-wide total for read and write I/O with the individual I/O for each Oracle datafile. This allows us to quickly generate an alert report to tell us which files are having the most I/O activity. If we were judicious in placing important tables and indexes into separate tablespaces and datafiles, this report will tell us exactly which database objects are the most active.

Note that you can adjust the thresholds for the *rpt_hot_files.sql* script. You can set the threshold to 25 percent, 50 percent, or 75 percent, reporting on any files that exceed this threshold percentage of total read and write I/O.

This is a very important script and appears in the generic *statspack_alert.sql* script. It is critical that the DBA become aware whenever an Oracle datafile is consuming a disproportionate amount of disk I/O. The script below is somewhat complex, but it is worth your time to carefully examine it to understand the query. Lets examine the main steps of this SQL statement:

We select the individual I/O from *stats$filestatxs* and compare the value for each file to the total I/O as reported in *stats$sysstat*.

The WHERE clause determines when a file will be reported. You have the option of adjusting the reporting threshold by commenting out one of the three choices — 25 percent, 50 percent, or 75 percent — of the total I/O.

🖫 Rpt_hot_files.sql

It is highly recommended that the DBA run this STATSPACK report daily so the DBA can constantly monitor for hot datafiles. Below is a sample of the output from this script. Note how it identifies hot files on an hourly basis.

```
***********************************************************
This will identify any single file who's read I/O
is more than 50% of the total read I/O of the database.
***********************************************************

Yr. Mo Dy Hr. FILE_NAME READS PCT_OF_TOT
---------------- ----------------------------------- ---------- ------
----
2000-12-14 14 /u02/oradata/prodb1/bookd01.dbf 354 62
2000-12-14 15 /u02/oradata/prodb1/bookd01.dbf 123 63
2000-12-14 16 /u02/oradata/prodb1/bookd01.dbf 132 66
2000-12-14 20 /u02/oradata/prodb1/bookd01.dbf 124 65
2000-12-15 15 /u02/oradata/prodb1/bookd01.dbf 126 72
2001-01-05 09 /u02/oradata/prodb1/system01.dbf 180 63
2001-01-06 14 /u03/oradata/prodb1/perfstat.dbf 752 100
2001-01-06 15 /u02/oradata/prodb1/bookd01.dbf 968 69

***********************************************************
This will identify any single file who's write I/O
is more than 50% of the total write I/O of the database.
***********************************************************

Yr. Mo Dy Hr. FILE_NAME WRITES PCT_OF_TOT
---------------- ----------------------------------- --------- -------
---
2000-12-18 21 /u02/oradata/prodb1/bookd01.dbf 2654 58
2000-12-29 15 /u02/oradata/prodb1/bookd01.dbf 1095 49
```

When we know those data files that consume a disproportional amount of I/O, we can tune the I/O by moving the data files to other disks.

Conclusion

As we have noted, tuning disk I/O is the dingle most important Oracle tuning activity, and the vast majority of all

Oracle tuning has the goal of reducing the amount of disk access. Configuration of the data buffer pools and optimal file placement also make a huge difference in Oracle performance, and this paper demonstrates several important tools and techniques for reducing expensive disk access.

Measuring Data Segment Statistics

CHAPTER 4

Digging at the Segment Level : Performance Diagnosis Reaches A Deeper Level

Oracle 9i Release 2 provides a very useful way to find out performance metrics at the segment level, hitherto impossible, enabling DBAs to pin point problems to a specific segment.

Toolbox: Oracle 9i Release 2 RDBMS. No special tools needed.

User Expertise Required: Advanced DBA Skills.

The biggest problem faced by any Database Administrator (DBA) trying to diagnose a problem is the lack of system statistics at a very granular level. STATSPACK report gives a very detailed performance metrics profile but that is at the system level only. Although that provides enough information on the overall health of the system, it does not provide the DBA with the information on specific objects that experienced or contributed to the problem, especially in areas where the problems are storage- and data access-related. For example, a typical performance problem is caused by heavy buffer access activity that might be the result of a lopsided index or just plain data distribution in a table, producing a wait event called "buffer busy waits." The STATSPACK report or a peek into the *v$sysstat* view alerts the DBA that such an event occurred, but it does not indicate the specific object on which that event occurred, leaving the analysis in limbo. If that event occurs when the analysis is going on, then the exact segment can be ascertained from the P1 and P2 parameters of *v$session_wait* view. However, as it usually happens, the suffering sessions are either completed or blown out to conserve resources, and, thus, the evidence disappears along with them.

Oracle 9i Release 2 provides a set of very useful performance views that allow drilling down to the segment level, not just system level, even after the event has occurred. For instance when you see a lot of buffer busy wait events in a STATSPACK report, you can then drill down further to find out which segments contributed to this wait event. This enhances the problem solving process immensely as the segments can be identified without real-time monitoring and can then be tuned further.

This article will explore such instrumentation and will present to the reader — specifically the DBA troubleshooting the performance problem — the means necessary to find out the wait events on the specific segments.

Background / Overview

First, I will illustrate the methodology with a case study in tuning. I will start with basic tuning methodology in practice today using STATSPACK report and then accentuate the findings from the data collected from the new views. A typical STATSPACK report has the following lines:

```
                                                      Avg
                                          Total Wait  wait    Waits
Event               Waits   Timeouts   Time (s)   (ms)     /txn
. . . .
buffer busy waits     3400          0          30    8.8     11.2
. . . .
```

It shows that the buffer busy waits event occurred 3400 times. In order to tune the buffer busy waits, we could do a lot of things — we could increase the freelist groups and freelists of the segment, or we could rearrange the distribution of the rows in the table in such a way that the blocks are not repeatedly picked up at the same time from two different session. However, to do either of these, it's necessary to know the exact segment that to be tuned. The STATSPACK report does not tell us which objects contributed to the buffer busy waits event, and without the segment level information, the tuning cannot possibly continue. Traditionally, we would place event 10046 for each of the sessions and see all the wait events in the generated trace files, which tend to be extremely large. In a typical system, which may contain several hundred applications, this approach may not be feasible. Additionally if the applications connect through Multi Threaded Server, it becomes difficult to isolate single segment level problems even if trace analysis is possible.

This information is now obtained from the new performance view *v$segstat* and *v$segment_statistics*.

Setting the Statistics Levels

In order for Oracle to collect those statistics, you must have proper initialization parameters set in the instance. The parameter is *statistics_level* and is set in the *init.ora*. The good news is that this is modifiable via ALTER SYSTEM command and some underlying parameters are even modifiable via ALTER SESSION. This parameter can take three values:

BASIC: At this setting Oracle des not collect any stats. Although this is not recommended, you may decide to set this in a fine-tuned production system to save some overhead.

TYPICAL: This is the default value. In this setting, Oracle collects the following statistics.

- Buffer Cache - These statistics advise the DBA how to tune the multiple buffer pools. The statistics can also be collected by setting another parameter *db_cache_advice* independently using initialization file, stored parameter file, ALTER SYSTEM or ALTER SESSION. If it's independently set, that setting takes preference over the statistics level setting.
- Mean Time to Recover - These statistics help the DBA set an acceptable Mean Time to Recover (MTTR) setting, sometimes due to the requirements from Service Level Agreements with the users.
- Shared Pool Sizing - Oracle can provide valuable clues to size the shared pool effectively based on the usage and these statistics provide information on that.
- Segment Level Statistics - These statistics are collected at the segment level to help determine the wait events occurring at each segment. We are interested in these statistics.
- PGA Target - These statistics help tune the Program Global Area effectively based on the usage.
- Timed Statistics - This is an old concept. The timed statistics were enabled in earlier versions with the initialization parameter *timed_statistics*. However, the statistic was so useful that Oracle made it default with the setting of *statistic_level*. It can be set independently, too; and if set, overrides the *statistics_level* setting.

ALL: In this setting al the above statistics are collected as well as an additional two.

- Row Source Execution Stats - These statistics help tune the sql statements by storing the execution statistics with the parser. This can provide an extremely useful tool in the development stages.
- Timed OS Statistics - Along with the timed statistics, if the operating system permits it, Oracle can also collect timed stats from the host. Certain operating systems like Unix allow it. It too can be set independently; and if set, overrides the *statistics_level* setting.

If you set these via any of the three methods, Initialization File, ALTER SYSTEM or ALTER SESSION, you can find out the current setting by querying the view *v$statistics_level* as follows:

```
SELECT ACTIVATION_LEVEL, STATISTICS_NAME, SYSTEM_STATUS, SESSION_STATUS
FROM V$STATISTICS_LEVEL
ORDER BY ACTIVATION_LEVEL, STATISTICS_NAME;
```

The output is placed in *Listing 1*(http://www.dbazine.com/code/Listing1.txt).

So, set the *statistics_level* to TYPICAL either by ALTER SYSTEM or by an initialization parameter file. Do not forget to restart the database if you choose the latter.

Segment Level Statistics Collection

Now that we have set up the collection, let's examine what we can get from there. The main dynamic performance view that is populated is called *v$segstat*. Here is a description of the view.

Column Explanation

TS#	Tablespace Number, corresponds to the TS# column in SYS.TS$
OBJ#	The Object ID, which corresponds to the OBJECT_ID in SYS.DBA_OBJECTS
DATAOBJ#	It corresponds to the DATA_OBJECT_ID in SYS.DBA_OBJECTS
STATISTIC_NAME	The most important one, the name of the statistics we are interested in STATISTIC# A unique number to denote each statistics above. This is NOT the same as the V$SYSSTAT statistics number.
VALUE	The current value of that statistic. Please note the value is cumulative, just like the statistic values in V$SYSSTAT. If you drop the segment and recreate it, the value is reset.

As you can see, the columns are somewhat cryptic. Oracle provides another view called *v$segment_statistics* which is based on the above view. This view has a lot more columns and is more descriptive with respect to the object identification. In addition to columns like the main view, it also references the names of the tablespace, the object, and the owner etc. so that the user can quickly join the view with actual names.

However this view is a little slow. It's a better idea to get the *object_id* from the *dba_objects* and search based on that. Here is the description of the columns of the *v$segment_statistics* view that are not present in the *v$segstat* view. The other columns are the same as in *v$segstat.*

OWNER	The owner of the segment
OBJECT_NAME	The name of the segment
SUBOBJECT_NAME	If the above is a table with partition, each partition has separate statistics. The partition is referred to as sub-object.
TABLESPACE_NAME	Tablespace where the segment resides
OBJECT_TYPE	Type of the segment, TABLE, INDEX, MATERIALIZED VIEW, and so on.

To find out what all statistics are collected, you can check the view *v$segstat_name* which describes the statistic name and the number.

Examining Detailed Statistics

Now we will dive in to examine the actual statistics that we populate. Since it lets us examine stats for a specific object, we can query like the following:

```
SELECT STATISTIC_NAME, VALUE
FROM V$SEGMENT_STATISTICS
WHERE OWNER = 'SCOTT'
And OBJECT_NAME = 'SALES';
```

This provides an output similar to *Listing 2* (http://www.dbazine.com/code/Listing2.txt) Most of these wait events are self-descriptive. Once again, these are cumulative; so, the numbers go up as more operations continue on that segment. In addition, like any system level statistics, these statistics are deleted when the database is shutdown.

These segment level statistics break down the mystery surrounding the statistics collected from *v$sysstat* or from STATSPACK reports. When baffled with a number of wait events that have already happened, the DBA can fall back on these statistics to dig deeper and identify the exact segments that experienced these waits which in turn contributed to the overall system wide wait for that event.

Improvements

With these basics placed in already, let's try to improve the collection and reporting methods to further refine the performance examination. This can be done by creating our own view in the same line as that provided by Oracle but with a little enhancement. Examining the view definition of *v$segment_statistics*, we note that the view refers to an internal table called *x$ksolsfts*. This internal table has a very useful column - the time when the statistics were collected. This column, FTS_STMP, can be used to our advantage to provide further information on the wait events. A new view, called *segstat_with_time* is built from the definition of the *v$segment_statistic*, identical to it except for the inclusion of a new column called TIME_STAMP. The view creation script is provided in *Listing 3* (http://www.dbazine.com/code/Listing3.txt). The TIME_STAMP column can let you know if the statistics are stale and help you decide whether you should rely on them completely. The other important column this view adds is the INSTANCE_ID, which identifies the instance in a Real Application Cluster (RAC) environment. This view also takes away all but the most useful columns.

Case Study

The usefulness of the segment level statistics can be best illustrated by a case study. Here we will create a wait scenario and then diagnose that with the segment level statistics. In the process, we will discover the facilities brought forth by Oracle 9i Release 2 that were missing in earlier releases. Please note that although the case study simulated the problems as expected when tested by the author, it is not guaranteed to produce the same behavior elsewhere. However, it should be able to help the reader understand the methodology.

Our example system is of OLTP nature. We are seeing consistent performance degradation and the objective of the exercise is to identify the problem and eliminate it. We have taken STATSPACK reports and they show high waits for "buffer busy waits" event. However, since the report does not provide information on specific tables or indexes that experience these waits, we can't start the process of segment tuning. Under Oracle 9iR2 this is possible.

For the sake of demonstration, we have a table called SALES. The table is created as per the script in *Listing 4* (http://www.dbazine.com/code/Listing4.txt). We will initially populate the table using a script in *Listing 5* (http://www.dbazine.com/code/Listing5.txt). Examining closely the *Listing 5*, you will notice that the *customer_id* column values are loaded one bunch at a time, making the records of a particular *customer_id* concentrated in a few blocks. Therefore, during an update where the records are picked up in the customer id sequence, they will be very much likely to be picked from the same block by two different sessions. The test case transaction is described in *Listing 6* (http://www.dbazine.com/code/Listing6.txt), named *stress.sql.* This program, a simple PL/SQL script updates records with either the odd or even numbered *sales_trans_id* depending upon the parameter passed to it, for each *customer_id* from 1 to 60. This script is run from two different sessions.

The parameter passed is 1 from one session and 2 from the other, e.g. @stress 1. If the sessions are kicked off at the exact same time, both sessions will operate on the same *customer_id* but on different records due to the odd and even numbered *sales_trans_id* values, eliminating locking. However, both sessions will most likely try to update the records in the same block, because the records are arranged in the *customer_id* order and both the scripts access the records for the same *customer_id.* This will create a buffer busy waits scenario that we will identify and eliminate.

Once the table is loaded, execute a STATSPACK report collection. Typically in a production scenario, you would have enabled the jobs to run STATSPACK regularly. To collect the statistics, you would have to login as the STATSPACK user, usually PERFSTAT and issue a command EXECUTE STATSPACK.SNAP. This provides your baseline collection stats.

Now run the stress script from two different sessions, with parameter 1 in one session and 2 in other. For attaining the same time execution, kick them from a scheduler like cron in UNIX or AT command in Windows. After they are run, collect the STATSPACK statistics again by issuing EXECUTE STATSPACK.SNAP. To generate the report, run the script *spreport.sql* under $ORACLE_HOME/rdbms/admin directory which will ask you the *snap_id* for the collections. Give the *snap_ids* just before and after the stress script. An excerpt from the generated report has been provided in *Listing 7* (http://www.dbazine.com/code/Listing7.txt). Under the Section "Top 5 Timed Events", we note that "buffer busy waits" is one. The system waited 3378 times for 49 seconds, about 2.83% of all the waits times.

Armed with the information we have to unearth the segment that experienced this wait event. Before Oracle 9iR2, it was impossible. In 9iR2, if you have setup the statistics collection by specifying *statistic_level* initialization parameter, then it is trivial. You would issue the following query:

```
SELECT OWNER, OBJECT_TYPE, OBJECT_NAME, VALUE
FROM V$SEGMENT_STATISTICS
WHERE STATISTIC_NAME = 'buffer busy waits'
```

The result is something like this. Of course, you may see a lot more in your environment.

```
OWNER  OBJECT_TYPE OBJECT_NAME      VALUE
-----  ----------- ----------      ------
SCOTT  TABLE       SALES             3302
```

What we see here is the buffer busy waits were experienced by the table SALES owner by user SCOTT. The figure 3302 also roughly corresponded to the figure we obtained from the STATSPACK report. You immediately know that the problem lies in the table SCOTT.SALES. In Pre-9i Release2 Oracle databases, this information would have been impossible to get. In an actual production system, you would probably see a lot more tables with the buffer busy waits and the sum of all will correspond to the figure obtained from STATSPACK report. This gives the DBA ability to pin down the segment either that is a victim of a wait event or a creator of one and to take corrective action.

Solution

In the above example since we identified the offending segment, we will take corrective steps to fix the problem. If you notice the buffer busy waits were because two sessions were trying to update the same block at the same time. This can be easily solved by making the distribution more even. In addition, by making sure a block is less packed, we can reduce the likelihood that a block will become hot. As a solution, we will recreate the table with smaller *pctused* and larger *initrans* and *maxtrans* parameters. This will make the table less dense. The table creation script is provided in *Listing 8* (http://www.dbazine.com/code/Listing8.txt). Next, we will load the table in a different way as listed in *Listing 9* (http://www.dbazine.com/code/Listing9.txt). Examine the script closely. It loads the *customer_id* values one after another until the maximum of 60 is reached and the cycle is repeated. This type of loading eliminates the likelihood that a particular block will be chosen at the same time by two sessions if the customer_id is the same.

After this change, execute STATSPACK.SNAP again and note the value of VALUE in *v$segment_statistics* for the table SLAES. Since the value is cumulative, you will need a reference value to compare. Now run the *stress.sql* script from two sessions the same way before, with parameter 1 and 2. Finally, take STATSPACK reports again and see the buffer busy waits statistics. It will be much less. Now examine the *v$segment_statistics* view for the table SALES; it should be much less too.

Conclusion

Oracle 9i Release 2 provided one of the best tools a DBA can possibly have, to drill down to the segment level for analysis and diagnosis of wait events, even after the fact that the wait event was experienced. This tool goes a long way in performance related troubleshooting, which was impossible till this time.

Some common wait events like free buffer waits, etc. are not present in the *v$segstat*, Hopefully Oracle will provide them in the future releases. This is no doubt an important step in the direction where performance diagnosis becomes a little easier for the DBA community.

For more information here are some links to learn more: Oracle 9i Release 2 Manuals at http://otn.oracle.com/docs/products/oracle9i/doc_library/rel ease2/index.htm. Search on *v$segstat* or *v$segment_statistics* for more information.

Optimizing Oracle Physical Design

CHAPTER 5

Optimal Physical Database Design for Oracle8i

Introduction

Oracle8i adds a number of significant new features in the areas of indexing and space management along with major upgrades to the support for Index Organized Tables (IOT's). The paper presents performance figures on index compression and IOT's, and analyzes the performance and operational impact of online table reorganization, which is now supported for IOT's. A strong case is made for migrating certain types of table to IOT's though, as discussed, the change may not be transparent at the application level.

The paper also looks at temporary tables, and explains how they can both improve performance and reduce one specific type of application failure. The circumstances under which temporary tables should be used are detailed along with application changes that should be considered. Performance improvements achieved from both locally managed and transportable tablespaces are also presented, and the potential implications are explained.

Physical Database Design 101

Physical Database Design is a large and complex subject, but this section sets out to cover the issues that most commonly require consideration when planning the physical structure of an Oracle database.

Because of the increasing use of disk striping and storage array controllers, this paper assumes that I/O load balancing can be achieved without the direct involvement of the Oracle DBA and does not discuss the placement of the container files used to store an Oracle database. In this context it may be worth noting that journaled file systems such as the Veritas File System have been shown to yield better Oracle performance than the standard UNIX file system.

What is Physical Database Design?

Most of the design decisions that have to be made when creating an Oracle schema are logical rather than physical, and concern the logical definition of tables and their columns, and views and their columns. An Oracle DDL statement such as

```
create table STOCK ( PART# number not null primary key
, QUANTITY number not null
, LOCATION varchar2(20) not null
) tablespace DATA01 pctfree 15 pctused 0;
```

contains both logical and physical elements. The table and column definitions are logical and some knowledge of them will be required to write queries against the table whereas the space management clauses are purely physical and no knowledge of them is required to perform data operations against the table. By this argument the DDL statement:

```
create index STOCK_LOCATION on STOCK (LOCATION)
tablespace DATA01 pctfree 50;
```

is purely physical. It cannot affect the result of a query or DML operation against the table even though it may have a radical effect on the performance of the operation.

Database Block Structure

It is important to realize that rows in an Oracle table are almost always true variable length, and that the row length typically changes with each update. The most common exception is a table in which every column is of datatype CHAR or DATE as this data is stored fixed length. However a common use of the SQL UPDATE command is to replace a NULL value, and in this case the row is guaranteed to expand in length. To allow for row expansion, tables that are subject to updates should be created with a value for the storage parameter *pctfree* that leaves enough space in each block for foreseeable row expansion. The default value of 10 (which means that INSERT will leave every block 90% full or less) is rarely ideal and tables that are only subject to INSERT and DELETE should always specify *pctfree* 0 to optimize space allocation.

If random rather than bulk deletions are performed against the table then it is also worth considering the value for *pctused*. This specifies the point at which the block will again become available for data insertion. The default value is 40, meaning that blocks are available for row insertion when they are less than 40% full. A *freelist* mechanism identifies the blocks available for insertion, and *freelist* maintenance carries a CPU and disk I/O penalty. In general the lower the sum of *pctfree* and *pctused*, the less *freelist* maintenance will take place. As this sum approaches 99 (the maximum permitted value) a series of negative effects will be observed, and DBA's are strongly recommended to specify *pctused* 0 wherever possible. On the other hand if a row expands and there is no longer sufficient space for it in its original block, then the row is migrated to another block and this causes a performance penalty on any indexed retrieval of the row. For a table with both a significant update rate and a high indexed retrieval rate *pctfree* should be set relatively high as the increased efficiency of indexed retrieval will outweigh the penalty of a slightly larger table.

The block structure of B*tree indexes is broadly similar but not identical. In this case *pctfree* is used only during index creation, and leaves distributed free space in index leaf blocks. If new keys arrive with random values rather than always being higher than the current highest key, then index space management during DML operations can be all but eliminated by rebuilding the index at regular intervals with distributed free space. To allow for a doubling of index size, the index would be built with *pctfree* 50. If, on the other hand, every new index key is higher than the previous highest key then the index should be built specifying *pctfree* 0.

Block Size

There have been a number of papers at recent Oracle conferences describing the advantages of using a database block size greater than the traditional standard of 2048 bytes.

DBA's are strongly recommended to create Oracle databases with a block size of 8192 bytes except where there are compelling arguments for use of a different size. Larger block sizes reduce the number of spanned rows (rows that cannot fit in a single block) and save disk space in all but the smallest tables because less of the disk is used for the gaps at the end of each block. This disk saving in turn speeds up full table scans. Increasing the block size will also reduce the height of the tree for many indexes, and speed up index lookup.

Unstructured Data

Until Oracle8 the only mechanisms for storing large units of unstructured or encapsulated data were the LONG and LONG RAW datatypes. These have a number of functional disadvantages, and they also store the unstructured data inline in the row. This significantly slows full table scans, and can also cause long chains of row pieces that have to be navigated even during processing that requires access only to the structured data. In Oracle8 the LOB datatypes allow unstructured data to be stored a separate segment with its own storage parameters. This has significant performance and storage management benefits, but unfortunately converting a schema from LONG to LOB datatypes requires non-trivial code changes, and many development tools do not support the LOB datatype.

Freelists

When Oracle needs a new block into which to insert table data, it checks the table's freelist and takes the block at the head of the list. If there are no blocks on the freelist it advances the high water mark (HWM), which records the last block which has ever contained data. If there are no blocks left beyond the high water mark then more space must be allocated to the table. This mechanism works well for small numbers of users inserting into the same table, but eventually the number of users sharing the same insert block causes serialization problems (they start having to queue to use the block). This can be detected by checking in *v$waitstat* for buffer busy waits on data blocks, and the solution is to recreate the object using the storage option freelists to add additional freelists.

In an Oracle Parallel Server (OPS) environment, tables that will be subject to inserts from more than one instance should be created with the storage option freelist groups to ensure that database instances do not have to share insert blocks. Always bear in mind, however, that only applications specifically designed for a parallel server environment will give good performance within that environment. Although it is outside the scope of this paper, there is rather more to designing for OPS than simply remembering to use freelist groups.

Extents

In Oracle every data dictionary object that requires storage owns a storage segment; this in turn consists of one or more extents each of which is a group of logically contiguous database blocks. It is up to lower levels of software and device controllers to determine whether the blocks are physically contiguous. All data blocks are the same size, but extents may be any number of blocks up to the capacity of the data file (or raw device) in which the extent resides. Each segment must exist solely within a single tablespace, but it may extend across multiple data files or raw devices within that tablespace and may therefore extend across the entire disk domain directly accessible by the server.

Extent sizing can be specified at both the tablespace and segment level using the storage parameters initial, next and *pctincrease*. Although extents can be any size, it is strongly recommended that every extent in a tablespace should be the same size. This is best achieved by setting the default initial and next for the tablespace to the same value, setting *pctincrease* to zero, and never specifying these parameters at segment or object level. The result is that classic tablespace fragmentation becomes impossible, as every free extent in the tablespace should be either the same size as the requested extent or a multiple of it.

Many DBA's are concerned that this practice will cause some objects (segments) to have an excessive number of extents. This raises the interesting question as to how many extents might be regarded as excessive. Provided that extents are a multiple of the multiblock read count there is no evidence of any performance effect from having multiple extents other than the load of allocating and deallocating the extents. Using Oracle 8.1.5 under NT Workstation 4.0 on a 366Mhz Pentium II with 256Mb of RAM, extent allocation took about 12 msec and extent deallocation about 5 msec. Over the life of the average table this load is trivial even for 1,000 extents. Despite this, the approach of using uniform partition sizes is normally associated with an arrangement where tablespaces are grouped by segment size rather than by object association. This issue is discussed further under Transportable Tablespaces below.

Space management within the SYSTEM tablespace should be left entirely to Oracle, and no user objects should ever be created in this tablespace.

AutoExtension

When a segment requires a new extent, and there is no free extent in that tablespace that is equal to or greater than the number of blocks requested, then the user receives an error. This can be partially overcome by allowing at least one of the files comprising the tablespace to autoextend. If this property is set then Oracle tries to enlarge the file by a specified amount until the file either exhausts the space available in that file system, or reaches a preset maximum length. This mechanism is highly valued by some, and totally deprecated by others.

Where a mount point or file system contains data files for many tablespaces, and the DBA is unable to predict which of these will run out of space first, then there may be some benefit in allowing the individual tablespaces to compete for the remaining space. However it is recommended that adequate free space should be preallocated to each tablespace used by any mission critical application, and that active monitoring be performed to predict space exhaustion before it occurs.

Partitioning

Oracle is entirely capable of managing tables of several hundred gigabytes, comprising hundreds of millions of rows. Performing maintenance operations such as bulk deletion, backup or index creation on such tables is challenging, especially in environments where maintenance windows are restricted. The requirement to have each segment within a single tablespace means that there must be a tablespace at least as large as the largest segment, and this poses real space management problems on most platforms.

The solution is to partition the logical object into many physical segments, splitting it up on the basis of a partition key comprised of one (or more) table columns. In Oracle 8.1 this can be done on the basis of key ranges or by a hash value based on the key. For exceptionally large tables it may make sense to first divide the table into a series of key ranges, often based on date, and then to subdivide these key ranges using hash partitioning.

Both tables and indexes can be partitioned, and an important feature of partitioning is that although every partition of an object must have the same logical structure, they may have different physical segment properties. Thus the bulk of the partitions of a history table can be directed to read only tablespaces on the grounds that past history may not be updated, whereas more current records can be placed in tablespaces that are available for writing. This approach can dramatically reduce regular backup times and backup volume.

Partitions, and especially date-based partitions, also offer highly efficient bulk deletion to partition-aware applications through the SQL DDL statement:

```
alter table … drop partition …;
```

This is especially attractive for partitioned tables with locally partitioned indexes. These are indexes where each index partition refers to one and only one table partition. This arrangement allows table partitions to be dropped and new table partitions to be added without any need to maintain a table-level index. The downside of this arrangement in OLTP applications is that unless the index key contains the partition key, then on index lookup every index partition must be visited. For a unique key lookup on a table with 1,000 partitions this would incur an overhead of several hundred to one when compared with a lookup on a global index on the same unique key (whether or not this global index was partitioned).

Global indexes can take considerable time (and enormous amounts of temporary segment space) to build, and become invalid or unavailable if any partition is removed or is inaccessible. However they offer the only efficient means of retrieving low numbers of rows from a very large table when the partition key is not among the criteria.

Index Compression

The bottom level of any B*tree index is the sequence set, an ordered list containing each key value with a pointer to the row that contains the key. In all previous versions of Oracle this "pointer" has been the rowid, though in Oracle8i the special case of Index Organized Tables requires a rather different convention, discussed later in this paper.

Although this ordered list was highly compressed in Oracle Version 5, more recent versions have stored every instance of every key in full and this can consume significant disk space. Oracle8i allows indexes with concatenated keys to be built with compression on a specified number of leading key columns e.g.

```
create index SAMPLE_WORDS
on SAMPLE (WORD1, WORD2, WORD3, WORD4, WORD5)
nologging compress 3;
```

In the testing performed for this paper, compression was always allowed to default to the maximum number of columns permitted (which in turn depends on whether or not the key is unique).

The author's experience of compressed indexes was almost universally positive. They saved significant amounts of space and were slightly faster to create than their uncompressed equivalent, presumably because there were fewer blocks to write. No significant performance differences were measured retrieving from compressed and uncompressed indexes, although time did not permit the testing of long index range scans. These were expected to favor compressed indexes because less index blocks would require to be visited.

A deliberately severe update test resulted in a 46% increase in CPU activity over the same test when applied to a table with an uncompressed index, but the increase in elapsed time was almost insignificant. This test involved updating the 3rd column of a 5 column compressed index, forcing the index entry to be deleted and moved to another part of the sequence set. No I/O penalty could be detected during this test.

Index Organized Tables (IOT's)

Tables of Organization Index were introduced with Oracle8, but had a number of restrictions that made them generally unattractive. In Release 8.1 most of the restrictions have been removed, and this special type of table looks to have become a realistic design option. The DDL to create them is straightforward, e.g.

```
create table SAMPLE6
( ID#
, constraint SAMPLE6PK primary key (ID#)
, CODE
, …
, SUBCODE
) organization index pctthreshold 20;
```

Put at its simplest, an IOT is a primary key index acting as a table. If you look in Oracle's online data dictionary, the table exists in *sys.tab$* but it has no matching entry in *sys.seg$*. An index segment, with the same name as the primary key constraint, is used to store the "table". For this to be effective the sequence set of the index has to be capable of storing non-key columns along with the key columns. As a result no entry in the sequence set may exceed half the block length. This restriction is required because a B*tree must be able to hold a minimum of two keys per sequence set block. When defining an IOT the user may specify the maximum sequence set entry size as a percentage of the available space in each block - the default maximum is 50%. Any data over this size (*pctthreshold*) is stored in a separate overflow segment.

The claimed advantages of IOT's are space savings (the primary key is only held once) and faster access because having located the sequence set entry Oracle has also located all of the column data unless that data is in an overflow segment. An IOT also breaks one of the "golden rules" of relational data by storing the data in a guaranteed order though it is risky for an application to rely on this property. If the application requires data in a specified order than that data should be retrieved using an ORDER BY clause.

With Oracle8i IOT's may have secondary indexes, but a potential problem arises here. The table rows are sequence set entries, and their position can and will change as other keys are added and deleted around them. Oracle has implemented a simple solution to this problem - a secondary index on an IOT stores the primary key of the target row rather than its rowid. Optionally a pseudo rowid may also be stored to allow more direct navigation to the target row, though over time this can become inaccurate and the navigation will revert to using the primary key.

Insert Times

Because the index structure has to be built during row insertion, and the rows must be correctly positioned in the sequence set, it has always been clear that inserting into an unindexed conventional table will be faster than loading into an IOT. On the other hand, tables of any size normally have at least a primary key index and therefore the total time to insert rows into the table must include the creation or maintenance of this index.

Tests were performed to compare the insertion of 100,000 rows into an IOT with the insertion of the same data into a conventional table followed by applying a primary key constraint to build an index. In the first test the primary key was long (WORD1, WORD2, WORD3, WORD4, WORD5) and although the conventional table loaded much faster, this advantage was lost in the time taken to build the index. The space saving was, as expected, massive.

TABLE TYPE	INSERT TIME	INDEX TIME	TOTAL TIME	TABLE BLOCKS	INDEX BLOCKS	TOTAL BLOCKS
Conven-ional	11	85	96	986	768	1,754
IOT	86	-	86	-	1,040	1,040

Table 1 *Performance with long primary key (all times in seconds).*

When a very short key was used on exactly the same data, a rather different picture emerged. The figures for the IOT barely changed at all, the conventional table was much faster to index, and its space overhead was reduced. The only way found to markedly reduce the insert time for the IOT was to present the data in key order, which removed the need to insert keys into the middle of sequence set blocks. Even in this case the insert time for the IOT was about 40% longer than the sum of the insert and index times for the conventional table.

TABLE TYPE	INSERT TIME	INDEX TIME	TOTAL TIME	TABLE BLOCKS	INDEX BLOCKS	TOTAL BLOCKS
Conven-tional	11	32	43	986	294	1,280
IOT	85	-	85	-	1,040	1,040
IOT (in key order)	60	-	60	-	1,040	1,040

Table 2 *Performance with short primary key (all times in seconds).*

Retrieval Times

The results from the retrieval tests were less marked than had been anticipated. Retrieval by primary key from an IOT with no overflow segment was faster than retrieval from the equivalent conventional table with a primary key index, and retrieval through a secondary index on an IOT was slower than retrieval through a secondary index on a conventional table. The differences in I/O traffic (more correctly block visits) were consistent with this model.

The overhead of using the secondary index on an IOT was not excessive. However it was felt that because retrieval via a non-unique index typically leads to more rows being retrieved per query than using a primary key index, retrieval from an IOT via a secondary indexes might amplify the negative performance impact in a production environment.

Application Impact

For almost all purposes an IOT has precise functional equivalence to a traditional Oracle table, now referred to as a Heap Organized Table. However no table used by existing applications should be converted to an IOT without first checking whether and how that application use the pseudo column rowid.. Represented in character form, the rowid for an IOT not only requires more space (42 bytes as against 18 for a conventional table) but also changes if the primary key is updated. This latter behavior will only affect an application that updates twice using the same rowid, once to update the primary key and then again to update any column in the row. Such behavior is deprecated.

Online Table Reorganization

With version 8.1 it is possible to rebuild or reorganize an Index Organized Table in parallel with normal use of the table, including DML. The minimal version of the syntax is delightfully simple e.g.

```
alter table SAMPLE move online;
```

The keyword online is optional. If it is not present then DML against the table is blocked for the duration of the operation but during timing tests it was found that in the absence of any update traffic move online was consistently around 30% faster than plain move. The reasons for this anomaly are unknown.

The great concern was that the time to perform the move, or rebuild, operation would be greatly increased if updates occurred in parallel with the alter table … move online but tests showed that the performance was surprisingly good. A compute bound update loop was coded in PL/SQL to perform 10,000 updates and commit after each update. Run on an otherwise empty machine this took 74 seconds to complete. The rebuild, run without any other load, took 21 seconds for a total of 91 seconds to perform the two tasks serially.

When the two tasks were run in parallel by starting the update and then immediately starting the rebuild, the total elapsed time for the update was 127 seconds with the rebuild running in parallel for 119 of those seconds.

A series of further tests were performed at lower update volumes. These demonstrated that updating in parallel with a rebuild or move approximately doubled the time taken per update and slowed the rebuild by almost exactly the time taken by the update. From the tests performed alter table … move online appears to impose a reasonable overhead and to scale well.

As stated above, these update tests were performed in transactions that contained only a single update. Execution of alter table … move online requires two "quiet points". Execution of the statement will neither start nor complete while there are uncommitted transactions against the table; while it is underway any number of transactions may be initiated against the table, but they must all complete before the rebuild will end. This is unlikely to be of concern unless the application contains very long running transactions.

For applications that want to get closer to 24X365 but expect to have a need to move or reorganize tables, the availability of alter table … move online presents a further motivation to consider the use of Index Organized Tables.

Temporary Tables

Many applications use the database to handle arrays of transient working data. This is especially common in applications written using Rapid Application Development (RAD) tools that may not feature the robust memory management features required in order to handle data arrays.

The use of fully persistent database objects (tables) to hold transient session-specific data can cause a number of performance and functional problems in an Oracle environment. All changes to such tables are recorded in both rollback segments and the redo log for recovery purposes, and in multi-user environments which share a permanent table for transient use it may be necessary to index every row inserted using the session id (SID). Shared tables also require the application to perform expensive deletes at end of transaction or session, and to implement recovery code to clean up after failed sessions. An alternative approach is for each session to create its own "temporary table" but there are a number of performance implications in this approach, and it does not scale well.

Oracle 8.1 introduces the statement form:

```
create global temporary table GTEMP
(COL1 …
, …
) on commit preserve/delete rows;
```

Any row inserted into such a table is visible only within the transaction that inserted it unless the qualifier on commit preserve rows is present. In this case the row continues to be visible to the creating session after commit and until deleted (or end of session). Global temporary tables are therefore a valid design option for any table whose data is never required to be persistent beyond the end of the transaction or session that inserted it.

These temporary tables are allocated in the current user's temporary tablespace, and observation indicates that a single temporary segment is used in each temporary tablespace for all temporary tables for all sessions that use that temporary tablespace. The temporary segment was not observed to shrink as temporary rows were automatically deleted at end of transaction or end of session, but it was apparent that the space released was being made available to new transactions. The only control over the tablespace used for temporary tables is through the SQL statement

```
alter user … temporary tablespace …;
```

Bulk insertion into an unindexed temporary table was about twice as fast as into the equivalent persistent table, and generated about 95% less redo log entries. More surprising, full table scans appeared to be about 25% faster. Temporary tables may be indexed, but this was not tested during the writing of this paper. It should be noted that DDL may not be performed on a temporary table if it contains rows for any session.

Application Impact

Many applications that use persistent tables as session workspace are subject to error conditions when the processing inadvertently bypasses the removal of entries made in the table. This problem can exist in any application that preserves the temporary data across more than one transaction, as the data will persist unless specifically deleted. The problem may be especially severe where the session creates its own table to use as workspace, because automatic transaction rollback on error will never undo a DDL operation. The result in both cases is that application failure can leave "temporary" rows in the table, and these may cause future sessions using the table to get incorrect results. The author has seen a number of applications that contained defensive code to recover from rows accidentally left by an earlier session. A major advantage of global temporary tables is that this scenario cannot occur.

In most cases an application can have a work table switched from a conventional table to a global temporary table without code changes being required. This should result in some performance improvement, and will remove the risk that a failed session will leave persistent rows in the database. However most applications will require modification to fully leverage global temporary tables. The changes are typically quite straightforward because they are mainly concerned with the removal of logic that is no longer required. Examples of functionality that can be removed include table create and drop for applications that built a table with a unique name per session, row deletion at end of transaction or session, daemons to tidy up after sessions that have failed to delete their rows, and indexing by session identifier. None of these are required when using global temporary tables.

Locally Managed Tablespaces

Conventional Oracle tablespaces have their space allocation recorded in the data dictionary in the tables *sys.fet$* and *sys.uet$*. Oracle 8.1 introduces an alternative, which is for the tablespace to contain an allocation bitmap for a series of equal size extents. The statement

```
create tablespace SAMPLE
datafile 'D:\oracle\oradata\…' size 1000M
extent management local uniform size 100K;
```

creates a 1 gigabyte tablespace in which every extent will be 100k bytes. Objects can be created specifying extent sizing, but these parts of the DDL are quietly ignored. For DBA's who like the idea of tablespaces with consistent extent sizes to prevent fragmentation, locally managed tablespaces provide the mechanism to fully enforce this approach. Unfortunately they also invalidate any space usage reporting scripts that rely on *sys.fet$* and *sys.uet$*. Such scripts must be extended to use *sys.x$ktfbfe* and *sys.x$ktfbue* for locally managed tablespaces. The columns in these virtual tables map easily to the column names in the equivalent data dictionary tables, and the additional virtual table *sys.x$ktfbhc* summarizes free space with one row per datafile. There are no *gv$* or *v$* views to externalize these virtual tables.

In a series of tests Locally Managed Tablespaces appeared to work well and to use significantly less resource for space allocation and deallocation. Traditional tablespaces were found to take more than three times as long to allocate an extent, and more than twice as long to deallocate. Impressive though these figures are, they are only significant to applications that perform altogether too many space management operations (possibly the creation and dropping of temporary tables). Nevertheless the improvements should be of major benefit to instances suffering from type ST lock conflicts. The author would argue that these can invariably be resolved by application design change, but this option is not available to many (most?) system administrators.

Transportable Tablespaces

It is a common requirement to migrate data from one Oracle database to another. Two principal mechanisms have been used in the past. Either the data was transferred using Oracle's distributed database support or it was unloaded from one database, a file physically transported to the destination, and the data reloaded. Both approaches have strengths and weaknesses, and both consume considerable resources and take significant processing time for large tables. In Oracle 8.1 one or more tablespaces may be copied and "plugged into" another database subject to a set of restrictions; in addition a single physical copy of a read only tablespace may be simultaneously part of many physical databases. One of the major restrictions in both cases is that the source and destination instances must be running on the same hardware and OS platforms, and must be the same Oracle release.

Although not strictly part of space management, transportable tablespaces have a number of implications for space planning in an Oracle 8.1 environment because the mechanism requires self-contained sets of tablespaces. Put simply, these are sets of tablespaces where all of the partitions and indexes are present for every table or cluster in the tablespaces. Optionally the definition may be extended to include all targets of referential integrity constraints. This is turn draws into question the use of specific tablespaces for a size of object, and makes more attractive the traditional approach of allocating tables and indexes to tablespaces on a functional basis.

The portable tablespace mechanism requires that the source tablespaces be read-only for the duration of the data copy phase of the operation, and therefore the source instance cannot be said to maintain 100% availability. Nonetheless with careful allocation of objects to tablespaces, transportable tablespaces can offer a highly efficient data transfer mechanism. They also provide an additional incentive for use of the autoextend mechanism to avoid the need to preallocate space to a tablespace, because unused datablocks within a tablespace simply add to the time taken to copy the datafiles and therefore to the time for which the tablespace must be read-only.

In a simple experiment under NT 4.0 on a 366 MHz Pentium II moving a tablespace containing 18 objects, the export processing on the source database took under 20 seconds, and the processing required to plug the tablespace into the destination database took less than 5 seconds. Of course copy time varies with tablespace size and hardware speed.

The major negatives found with the feature were that it was liable to "finger trouble" when transporting multiple tablespaces, and that it was necessary to enter the SYS password from the keyboard on each instance in order to operate the mechanism.

Conclusions

To summarize the key recommendations made in this paper:

- Previous good practice in space management remains valid with Oracle 8.1, especially the notion of equal extent sizes throughout a tablespace. Locally managed tablespaces allow this approach to be enforced and have performance advantages though few sites should be performing enough space allocation to see a significant change in overall performance as a result.
- Database block size should be 8192 bytes in most cases, and almost never 2048 bytes.
- If possible LOB datatypes should be used in place of LONG datatypes.

- Care in selecting the values of *pctfree* and *pctused* for tables, and *pctfree* for indexes, will be rewarded by improved performance and reduced need to reorganize.
- Index compression is recommended for concatenated indexes as a way of both reducing index create time and saving space.
- Index Organized Tables are worth considering for any large table whose primary key is a substantial part of the average row, and for tables that may benefit from online reorganization. With either of these conditions satisfied, the less secondary indexes are used to access the data, the stronger the motivation to use an IOT.
- If applications must use the database as working storage, then global temporary tables have significant performance advantages and will allow the application to be simplified and made more robust in many cases.
- Portable tablespaces look to be a promising feature, but one that may require reappraisal of the way in which objects are allocated to tablespaces.
- The majority of these recommendations have no impact whatever on the application code, and all have the potential to improve application performance.

CHAPTER 6

Verifying Segment Backup Scripts

Did the Backup Work?

We have clients set up with a mixture of hot and cold backups. The question asked each morning is 'Did the backup work?' Right now we are telnet--ing to the server and checking the log file generated by the backup script, or we have this log file emailed to the support DBA so the check can be done without telnet. This is fine, but we have to connect into the instance each morning anyway for all of our normal morning checks. Could we combine these multiple check actions into one? I would rather see all from inside the instance.

Problem

I want to bring information from outside of the instance, into the instance. The restriction is I don't want to write anything in C or any other language that would not normally be supported. So I can use SQLPlus and shell scripts only in order for this to all be supportable by the next DBA. There are always several possible approaches. Here is one that I think will be easily supported and understood.

Our requirements are:

- Know if the file copy completed successfully
- Know when the file copies were performed last
- Keep all of this inside of the instance

- Have this information available to anyone

How Do We Know?

The most common way to know that the backup worked correctly is to view the log file. But what if we don't want to wade through the log file or set up *grep* scripts to pull out just the lines we want? I am more a fan of just doing the *ls -l* command on the backup mount point. This shows me the file size and the date modified. So I expect this to be the same as the data mount points. Here I have to admit to a certain laziness. I don't always check the file size against the source size. I just check that there is some free space on the backup mount point and that the file date is from last night. This really is not secure enough. So let's work with the *ls -l* command since that will give us the most secure results if we do it correctly.

```
> ls -l *.dbf
-rw-r-----   1 oracle   dba      1468022784 Jul 30 03:16 data0101.dbf
-rw-r-----   1 oracle   dba      115359744 Jul 30 03:17
data_stage0101.dbf
-rw-r-----   1 oracle   dba      68173824 Jul 30 03:18 history0101.dbf
. . .
```

Parsing This String

The information we are really interested in is, the file name, the time and date modified, and the size in bytes. If we had a line of data already in a variable in the instance we could parse out the information we want.

If we are working from the right to the left, we know that the file name is everything at the right end of the line, up to the blank. So we can pull out the file name and then remove it from the line.

```
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_name from dual;
str := rtrim(substr(str,1,blnk));
```

Now we have the time at the end of the line. We can take that off using the same approach and do the same for date, month and size to get the rest of the data we want.

Then to save this information we will create a table:

```
create table t_sizes
(file_name   varchar2(60)
,dt          date
,bytes       number(12))
tablespace meta_data storage (initial 64k next 64k pctincrease 0)
pctfree 0 pctused 80;
```

Now we can put this data into the table so it can be viewed by anyone at anytime.

```
insert into t_sizes (file_name,dt,bytes)
values (file_name
        ,to_date(file_mon||' '||file_date||' '||file_time,'mon dd
hh24:mi')
       ,file_size);
```

Bring It In

So we have the code that can be used to parse out the line, but how do we get that line in? We chose the following looping structure:

```
while read PAR1
do
sqlplus id/pwd <<EOF
declare str varchar2(180) := '$PAR1';
begin
. . .
end;
/
exit
EOF
done
exit 0
```

This allows us to treat each line as a parameter to the SQLPlus script. The shell 'while' loop gets the next line and serves it to the script. So if we pipe the output of the *ls -l* command into this loop structure, we will process each line and put the data into a table. We have the following as a driver (in a file called *sizes.sh*):

```
sqlplus id/pwd <<EOF1
truncate table t_sizes; -- we only want the current information
exit
EOF1
ls -l /oraback/*.dbf | sizes_load.sh
```

The file *sizes_load.sh* then contains:

```
while read PAR1
do
sqlplus id/pwd <<EOF
declare str varchar2(180) := '$PAR1';
blnk number(2);
file_name varchar2(60);
file_time varchar2(5);
file_date varchar2(2);
file_mon varchar2(6);
file_size varchar2(12);
begin
str := rtrim(ltrim(str));
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_name from dual;
str := rtrim(substr(str,1,blnk));
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_time from dual;
str := rtrim(substr(str,1,blnk));
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_date from dual;
str := rtrim(substr(str,1,blnk));
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_mon from dual;
str := rtrim(substr(str,1,blnk));
select instr(str,' ',-1,1) into blnk from dual;
select substr(str,blnk+1,60) into file_size from dual;
str := rtrim(substr(str,1,blnk));
insert into t_sizes (file_name,dt,bytes)
values (file_name
       ,to_date(file_mon||' '||file_date||' '||file_time,'mon dd
hh24:mi')
       ,file_size);
commit;
end;
/
```

```
exit
EOF
done
exit 0
```

These scripts are run each night at the end of the backup script so we have the most current data in the instance.

Use It

Now that we have this data in a usable form in the instance, how do we check that the backup was ok? What we want to do is just check these actual file sizes against the file sizes defined in *dba_data_files*. When I did this the first time my script told me that there was a difference in the file size for all data files. I then saw the same difference in the size defined by *dba_data_files* and the actual live data files. The file size in *dba_data_files* is one block, as defined by *dba_block_size*, less than the actual size. I assume this is the file header put on by the instance. So our comparison code becomes:

```
variable block_size number
begin
select value into :block_size from v$parameter where name =
'db_block_size';
end;
/

select a.name,a.bytes dba_bytes,b.bytes backup_bytes,b.bytes-a.bytes
diff,b.dt file_date
from
(select substr(file_name,instr(file_name,'/',-1,1)+1,30) name,bytes
from dba_data_files) a
,(select substr(file_name,instr(file_name,'/',-1,1)+1,30) name
 ,bytes-:block_size bytes,dt from t_sizes) b
where a.name = b.name(+);
```

```
NAME                                DBA_BYTES BACKUP_BYTES       DIFF
FILE_DATE
------------------------------ ---------- ------------ ---------- -----
----
system.dbf                           81788928     81788928          0 30-
JUL-02
patrol_data01.dbf                    31457280     31457280          0 30-
JUL-02
data0101.dbf                       1468006400   1468006400          0 30-
JUL-02
```

So we can see that the files were created this morning and the sizes match. Our backups worked wonderfully!

Use it Elsewhere

There are other ways to bring this data in but this is one that works for us. We are also using this same approach to bring in other information, like free space on the mountpoints.

Now I can perform all of my morning checks with one script inside of the instance.

CHAPTER 7

Data Segment Update Internals

How Much Does an Update Cost?

There are several application generators on the market that adopt the basic approach of "keep it simple." Simple code can be easier to generate and easier to maintain, even if it may seem a little less efficient. But if a screen generator always generates code to update every column in a table, even if the user changes just one field on the screen, how much does this cost you at the database?

A Brief History of Screen Generators

Once upon a time, SQL*Forms (as it then was) used to have a single SQL statement embedded for each block that was that block's update statement. This statement would update (by rowid) every column in the table that had been referenced in the block. This seemed to be a nice idea, because it kept the code simple and efficient at the client end of the system: there was no need to run a CPU-intensive task to discover which fields had actually changed, and no need to construct dynamically an exact piece of SQL to update the matching columns at the database.

Some time around Forms 4.5 (I may be wrong with the version, and wait to be corrected), Oracle introduced a flag that you could set to make the choice between a single 'update everything' statement, and a dynamically generated 'update just the minimum set of columns' statement. Which option is the smarter?

What Does It Cost to Update a Column?

Specifically we are interested in what it costs to update a column without changing it. If the database could detect that an incoming update was not actually going to change anything then the marginal cost of the redundant update would be minimal. Unfortunately, the database does not try to check for redundant updates. After all, it's reasonable to assume that updates are supposed to change the data. It would be counter-productive to add a check that was almost always true simply to make a small saving on those 'extremely rare and pointless' occasions when a 'no-change' update arrives.

So what could happen if you update a single column in a table using a single transaction? Clearly, the row has to be locked and the data modified, so an Interested Transaction List (ITL) entry in the block has to be acquired. A transaction table slot has to be taken in the undo segment header to act as a globally visible "reference" for the transaction, and an undo record has to be written into an undo block to describe how to reverse out the changes you have just made to the data block. The changes to all three blocks have to be recorded in the redo log (initially in just the log buffer), and in this simple case this will take just one redo record.

When you then commit the transaction, the transaction table slot is updated with the commit SCN and marked as free, and the address of the undo block you have used may also be written back into the free block pool in the undo segment header block. These changes to the undo segment header block are recorded in the redo log (buffer) and the log writer process (lgwr) is called to flush the log buffer to disc, after which your process is informed that the commit has succeeded. (Oracle may also, and in this case probably would, go back and clean up the changed data block, but would not record these clean-up changes in the redo log).

Assume that you, the user, updated just one field on screen — the description above covers the amount of work the database has to do to apply your change. But what if the screen generator has updated the whole row — what marginal costs appear?

The cost of taking the ITL entry, locking and updating the row has probably not changed much, and the cost of acquiring the undo segment header block has not really changed.

But the volume of data written to the undo record has probably gone up significantly — instead of an overhead of about 100 bytes plus the old version of one column, we now have the overhead plus the old version of all the columns. But perhaps that won't matter too much, after all Oracle writes in blocks not records — but if our undo records are now four times the size we will, on average, write about four times as many undo blocks as we need to.

Similarly, the redo record will have changed significantly. In the perfect case, the redo record would have been about 200 bytes long for the update (plus a further 200 relating to the segment header block and transaction audit vector). This would probably have been padded to 512 bytes (the typical o/s block boundary) with redo wastage as we issued the commit. But the main redo record consists largely of two change vectors — the vector for the table block, and the vector for the undo record

— and both of these have increased in size because the undo's redo record now includes the old version of all the columns and the table's redo record now includes the new version for all the columns. So the rate at which you churn out redo may have gone up by a factor of something between two and four — and in many high-throughput systems the speed of getting the redo to disc is often a critical performance issue.

But There's More

Extra volume of undo and redo isn't necessarily the most significant issue though — after all, the extra volume generated and buffered isn't usually dramatic given the size of the basic overheads. Furthermore, because Oracle's basic architecture tries to push disk writes into background "asynchronous" processes, the end-user doesn't often see the time-lag due to disk-writes. But there are several considerations that will have a direct impact on the way that the end user sees the performance of the system.

In the event of a column being updated with a 'no-change' update, what do you think Oracle does about:

Row-based triggers of the type 'update of {column list}'

- Before row
- After row
- Instead of

Updates to indexes that include that column

- What if those are B*tree indexes
- What about bitmap indexes
- What about function-based indexes

What does Oracle do about referential integrity

- If this is a column at the child end of a relationship
- If this is a column at the parent end of a relationship

Triggers

Create a simple table with trigger, and try a test like the one show in figure 1:

```
create table t2 (
    id_gp   number(4),
    id_p    number(4),
    n2              number(4)
);

insert into t2 values (1,1,1);
commit;

create or replace trigger t2_bru
before update of id_gp on t2
for each row
-- when (
--      new.id_gp != old.id_gp
--   or new.id_gp is null and old.id_gp is not null
--   or old.id_gp is null and new.id_gp is not null
-- )
begin
    dbms_output.put_line('Updating');
end;
/

column rid new_value m_rid

select rowid rid
from t2
where rownum = 1;

update t2 set id_gp = id_gp where rowid = '&m_rid';
```

The trigger fires. The same thing happens if the trigger is an after-row update. It is left as an exercise to the reader to confirm my assumption that the same thing happens on an instead of trigger.

In passing, a before-row trigger actually generates one extra undo record and one extra redo record — even if it takes no action because of a when clause, such as the necessarily complex clause commented out in the example — so if you have the choice, it is probably a little more efficient to use after-row triggers.

Indexes

You have to be a little fussier with some of the experiments with indexes, as you may want to count logical I/Os to get a full picture of the cost, and this may require you to build a table that is large enough to have a multi-level index. However, with some tests it will be sufficient to examine undo, redo, and locks, and not rely on something as sensitive as logical I/O.

So what happens if you do a 'no-change' update on a column which is part of a simple B-tree index? Nothing. Oracle detects that the indexed value has not changed, and doesn't even traverse the index, let alone lock an entry. The same is true of simple bitmap indexes.

You might wonder if something nasty happens when you switch to function-based indexes - does the function have to be called 'just in case', does Oracle track the dependency between the functions and the columns involved in the function properly? The answer is that everything works properly - you don't find redundant executions of the function or trips to the index. As a test case, you could start with the table from fig. 1, and run the following SQL.

```
 create or replace function my_fun(
           i1      in      number,
           i2      in      number
)return number deterministic
as
begin
    dbms_output.put_line('Testing function');
    return i1 + i2;
end;
/

create index t2_idx on t2(my_fun(id_gp,id_p));

update t2 set id_gp = id_gp where rowid = '&m_rid';

update t2 set id_gp = id_gp + 1 where rowid = '&m_rid';
```

You will find that the function is called once (because there is only one row in the table) as the index is created, but it is not called for the 'no-change' update. By the way, the function will be called twice in the second update - once to find the original location in the index, and once to calculate the new location. You may find that the function is called twice more if you issue a rollback - I believe I observed this in the earliest releases with function-based indexes - but it doesn't seem to happen any more.

Referential Integrity

This, perhaps, is where the crunch comes on OLTP systems. You get hit twice - first as a child, and then as a parent.

Take the table from fig. 1, and insert another 9, identical rows into it to get a total of 10 rows. Then run the following tests and check the logical I/O etc.:

```
update t2 set id_gp = id_gp;
> 10 rows updated.
alter table t1
add constraint t1_pk primary key (id_gp);
alter table t2
add constraint t2_fk_p1 foreign key (id_gp) references t1;
update t2 set id_gp = id_gp;
> 10 rows updated
```

You will find that the number of db block gets (current mode gets) goes up by 10 when the integrity constraint is in place. Why? Because on each update, Oracle checks the foreign key constraint - and it does this by tracking down the parent's primary key index using current mode gets. If you have a large parent table, this could mean three current mode gets every time you update a child column redundantly.

Switching to the parent end of the trap. You need only try to do a 'no-change' update on a parent row when there is no index on the foreign key on the child table to discover that you get the dreaded TM/4 lock. Foreign key indexes aren't necessary if you don't update or delete parent key values, but if you are having problems with random 'hangs' and deadlocks being reported then perhaps you have the classic problem, where you know you don't update the parent keys, but your application generator is doing it behind your back.

There's Always a Trade-off

By now, you may have decided that you obviously have to go back and rewrite lots of code. But writing code to produce the perfect SQL statement every time is likely to increase the risk of errors in the code. The trade-off between risk (and time to code, test and debug) and performance is always a valid point of argument, so if the server is nowhere near capacity then it may be perfectly sensible to ignore the issue — at least in the short term.

And there's another, more subtle, trade-off. If your application generates the perfect SQL for every update that the front-end can fire, then the number of different SQL statements could escalate dramatically.

In theory, if your table has N columns, then there are power(2,N) — 1 possible update statements — even if you restrict yourself to single row updates by rowid. Unless you increase the size of the shared pool, and tweak a couple of parameters such as *session_cached_cursors*, you may find that your savings in one area are offset by extra expenses (such as library cache contention) appearing elsewhere.

Conclusion

Allowing front-end tools to take the easy option when updating data - by writing a single SQL statement for all possible updates — can add a significant load to your system. If you are running client/server, or N-tier, then you may be better off using extra CPU at the client end of the system to build custom SQL to minimize the cost at the server. The decision is not black and white, though. Make sure that the cost is worth the benefit.

CHAPTER 8

Segment Transaction Slot Internals

Interested Transaction List (ITL) Waits Demystified

What is ITL?

Ever wondered how Oracle locks rows on behalf of transactions? In some RDBMS vendor implementations, a lock manager maintains information on which row is locked by which transaction. This works great in theory, but soon the lock manager becomes a single point of contention, as each transaction must wait to get a lock from the manager and then wait again to release the lock. This severely limits the scalability of the applications. In fact, application developers of some RDBMS products despise holding locks for a long time, and often resort to a full table lock when all that's needed is to get a few rows locked. This creates further waits, and consequently, scalability suffers.

So how is that different in Oracle? For starters, there is no lock manager. When a row is locked by a transaction, that information is placed in the block header where the row is located. When another transaction wishes to acquire the lock on the same row, it has to travel to the block containing the row anyway, and upon reaching the block, it can easily tell that the row is locked from the block header. There is no need to queue up for some single resource like a lock manager. This makes applications immensely scalable.

So, what portion of the block header contains information on locking? It is a simple data structure called "Interested

Transaction List" (ITL), a linked list data structure that maintains information on transaction address and rowid. ITL contains several *slots* or place holders for transactions. When a row in the block is locked for the first time, the transaction places a lock in one of the slots with the rowid of the row that is locked. In other words, the transaction makes it known that it is interested in the row (hence the name "Interested Transaction List"). When the same transaction or another one locks another row, the information is stored in another slot, and so on. After a transaction ends via commit or a rollback, the locks are released and so are the slots that were used to mark the blocks, and these newly freed slots are reused for the other transactions. So there is in fact a queue, but it's at a block level, not at the entire database level or even at a segment level.

The next logical question that comes up is, how many slots are typically available? During the table creation, the *initrans* parameter defines how many slots are initially created in the ITL. When the transactions exhaust all the available slots and a new transaction comes in to lock a row, the ITL grows to create another slot. The ITL can grow up to the number defined by the *maxtrans* parameter of the table, provided there is space in the block. Nevertheless, if there is no more room in the block, even if the *maxtrans* is high enough, the ITL cannot grow.

What Is an ITL Wait

So, what happens when a transaction does not find a free slot to place its lock information? This can occur because either (i) the block is so packed that the ITL cannot grow to create a free slot, or (ii) the *maxtrans* has already been reached. In this case, the transaction that needs to lock a row has to wait until a slot becomes available. This wait is termed as ITL waits and can be seen from the view *v$session_wait*, in which the session is waiting on an event named "enqueue."

Let's see this description of the wait in action. Assume our table has *initrans* of one and *maxtrans* 11. A typical data block right after the creation of the table will look like figure 1.

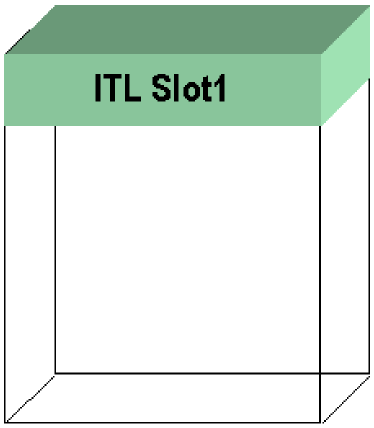


Figure 1

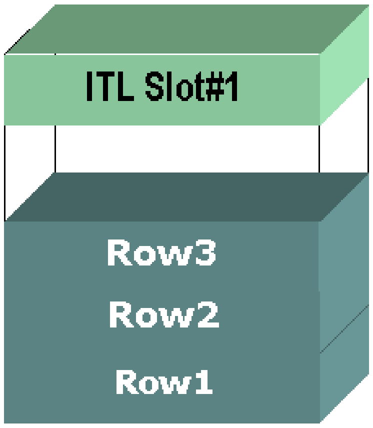


Figure 2

Since the *initrans* is one, there is only one slot for the ITL. The rest of the block is empty. Now we inserted three rows into the table. These will go into this block, and the block will look like figure 2.

Note how the empty space is reduced. At this point, a transaction called Txn1 updates Row1, but does not commit. This locks Row1, and the transaction places the lock in the slot number one in the ITL as shown in figure 3.

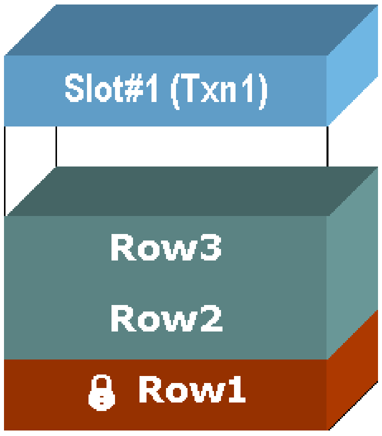


Figure 3

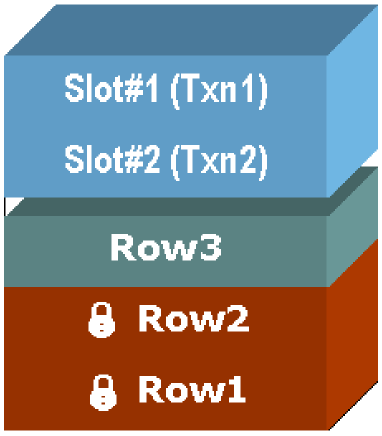


Figure 4

Then another transaction, Txn2, updates the row Row2 and wants to lock the row. However, there are no more slots in the ITL available to service the transaction. The *maxtrans* entry is 11, meaning the ITL can grow up to 11 slots and the block has empty space. Therefore, ITL can grow by another slot and Slot number two is created and allocated to Txn2 (refer to figure 4).

Now the empty space in the block is severely limited, and it will not be able to fit another ITL slot. If at this time another transaction comes in to update the row three, it must have a free slot in the ITL. The *maxtrans* is 11 and currently only two slots have been created, so another one is possible; but since there is no room in the block to grow, the slot can't be created. Therefore, the Txn3 has to wait until either of the other transactions rolls back or commits and the slot held by it becomes free. At this time the session will experience an ITL waits event as seen from the view *v$session_wait.*

Simulation

To better illustrate the concept, let's illustrate such waits using a case. Create the following table and then populate it with several rows. Note *maxtrans* value.

```
CREATE TABLE TAB1
(     COL1        NUMBER,
      COL2        VARCHAR2(200))
INITRANS 1 MAXTRANS 1
/
DECLARE
      I     NUMBER;
BEGIN
      FOR I IN 1..10000 LOOP
            INSERT INTO TAB1 VALUES
            (I,'SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS');
      END LOOP;
      COMMIT;
END;
/
```

Now update a row of the table from one session, but do not commit it.

```
UPDATE TAB1 SET COL2 = 'UPDATED' WHERE COL1 = 1;
```

From another session, update row number two and do not update it.

```
UPDATE TAB1 SET COL2 = 'UPDATED' WHERE COL1 = 2;
```

This session will wait. Why? It's updating a row for COL1 = 2, not the same row updated in the other session for COL1 = 1. So why is the session waiting? It's because the first transaction occupied the only available ITL slot. The second transaction needed another slot to place its lock information, but since the *maxtrans* I defined is one, the ITL could not grow to create another slot. Thus, the latter transaction has to wait until the former session releases the lock and makes the slot available.

Now increase the MAXTRANS of the table by issuing

```
ALTER TABLE TAB1 MAXTRANS 11;
```

and redo the above test. The second session will not wait this time because the ITL had enough free slots for both transactions.

How to Reduce ITL Waits

The primary cause of ITL waits is that free slots in the ITL are not available. This can be due to

- low setting of the *maxtrans*, which places a hard limit on the number of transactions that can have locks on a block
- or, the block is so packed that there is no room for the ITL to grow OR
- or both

Therefore, setting a high value of *initrans* will make sure that there are enough free slots in the ITL, and there will be minimal or no dynamic extension of the ITL. However, doing so also means that there is less space in the block for actual data, increasing wasted space.

The other option is to making sure the data is less packed so that ITL can grow enough to accommodate the surges in ITL. This can be done by increasing *pctfree*, increasing *freelists* and *freelist groups* parameters for a table. This will make a block hold less data and more room for the ITL to grow. As a direct result of the reduction in packing, the table will experience fewer buffer busy wait events, and performance will be increased.

How to Diagnose the ITL Wait

How do you know that a segment is experiencing ITL waits? The best answer will be found in the Segment Level Statistics provided in Oracle9i Release 2. To check for ITL waits, set up the *statistics_level* to TYPICAL in *init.ora* or via ALTER SYSTEM, then examine the segment statistics for the waits.

```
SELECT OWNER, OBJECT_NAME
FROM V$SEGMENT_STATISTICS
WHERE STATISTIC_NAME = 'ITL waits'
AND VALUE > 0
```

This unearths the objects that were subjected to ITL waits since the start up of the instance. Note that this view resets when the instance bounces. (For a more detailed explanation of this view and how to set it up, please refer to the article by this author here in DBAzine.)

In versions earlier than 9i, checking for ITL waits is tricky. When you suspect that a database is suffering from these waits, query the view *v$session_wait*. If the event on which the system is waiting is "enqueue," then the session might be experiencing ITL waits. However, enqueue is a very broad event that encompasses any type of locks, so it does not accurately specify the ITL waits. When the wait event is experienced, issue the following complex query:

```
Select s.sid            SID,
      s.serial#   Serial#,
      l.type            type,
      ' '         object_name,
      lmode             held,
      request     request
      from v$lock l, v$session s, v$process p
      where s.sid = l.sid and
            s.username <> ' ' and
            s.paddr = p.addr and
            l.type <> 'TM' and
            (l.type <> 'TX' or l.type = 'TX' and l.lmode <> 6)
```

```
union
select      s.sid       SID,
      s.serial#   Serial#,
      l.type            type,
      object_name object_name,
      lmode       held,
      request           request
      from v$lock l, v$session s, v$process p, sys.dba_objects o
      where s.sid = l.sid and
            o.object_id = l.id1 and
            l.type = 'TM' and
            s.username <> ' ' and
            s.paddr = p.addr
union
select      s.sid       SID,
      s.serial#   Serial#,
      l.type            type,
      '(Rollback='||rtrim(r.name)||')' object_name,
      lmode             held,
      request           request
      from v$lock l, v$session s, v$process p, v$rollname r
      where s.sid = l.sid and
            l.type = 'TX' and
            l.lmode = 6 and
            trunc(l.id1/65536) = r.usn and
            s.username <> ' ' and
            s.paddr = p.addr
order by 5, 6
/
```

The output of the query will look something like this.

```
 SID SERIAL# TY OBJECT_NAM       HELD  REQUEST
----- ------- -- ---------- ---------- --------
   36    8428 TX                     0        4
   36    8428 TM TAB1                3        0
   52   29592 TM TAB1                3        0
   52   29592 TX (Rollback=RBS1_6)   6        0
```

Note how the sessions 36 and 52 both have a TM (DML) lock on the table TAB1 of type 3 (Row Exclusive), but session 52 also holds a TX (Transaction) lock on the rollback segment of mode 6 (Exclusive) and Session 36 is waiting for a mode 4 (Share) lock. If this combination of locking occurs, you can be sure that session 36 is waiting for ITL on the table TAB1. Beware of a similar but different diagnosis when two sessions try to insert the same key value (a real locking – primary key violation). In that case, you would also see an additional TX lock on a rollback segment from the session that is waiting; for ITL waits, this TX lock on the rollback segment would not be seen.

Needless to say, this is a rather convoluted and inaccurate way to diagnose the ITL waits in pre-Oracle9i Release 2 versions.

What INITRANS Value is Optimal

Conversely, how do you know if the *initrans* setting is too high and the space is just being wasted? Ascertaining this is possible by using a few random block dumps from the segment in question. First, find out the header file# and header block# of the segment by issuing the following query:

```
SELECT HEADER_FILE, HEADER_BLOCK
FROM DBA_SEGMENTS
WHERE OWNER = '...'
AND SEGMENT_NAME = '...';
```

Use the output of the query to do a block dump of the header block.

```
ALTER SYSTEM DUMP DATAFILE <file#> BLOCK MIN <block#> BLOCK MAX
<block#>;
```

This will produce a trace file in the *user_dump_destination* directory. Open the trace file and find out the section on extent control via the following:

```
Extent Control Header
-----------------------------------------------------------------
Extent Header:: spare1: 0 spare2: 0 #extents: 1 #blocks: 10
last map 0x00000000 #maps: 0 offset: 2080
Highwater:: 0x02011f87 ext#: 0 blk#: 0 ext size: 10
#blocks in seg. hdr's freelists: 0
#blocks below: 0
mapblk 0x00000000 offset: 0
Unlocked
Map Header:: next 0x00000000 #extents: 1 obj#: 53689 flag: 0x40000000
Extent Map
-----------------------------------------------------------------
0x02011f87 length: 10
```

Find out the real number of blocks for the segment from *dba_segments* via the following:

```
SELECT BLOCKS FROM DBA_SEGMENTS
WHERE OWNER = '...' AND SEGMENT_NAME = '...';
```

Say this returns 12, and the #`blocks` shows 10; this means the first two blocks are header blocks; the data starts at the third block. Take a dump of the third block, which is obtained by adding two to the header `block`# obtained above.

```
ALTER SYSTEM DUMP DATAFILE <file#> BLOCK MIN <block#> BLOCK MAX
<block#>;
```

This will produce another trace file in the *user_dump_dest* directory. If you issued it during the same session as above, then the trace will be written in the trace file opened earlier. Open the file and locate the following section.

```
buffer tsn: 8 rdba: 0x02011f88 (8/73608)
scn: 0x0000.389b8d81 seq: 0x01 flg: 0x04 tail: 0x8d810601
frmt: 0x02 chkval: 0x2ef5 type: 0x06=trans data
Block header dump: 0x02011f88
Object id on Block? Y
seg/obj: 0xd1ad csc: 0x00.389b8d81 itc: 4 flg: - typ: 1 - DATA
fsl: 0 fnx: 0x0 ver: 0x01
Itl  Xid                 Uba                Flag Lck Scn/Fsc
0x01 0x0003.003.000024cc 0x00804067.050a.13 C-U- 0 scn 0x0000.389b304e
0x02 0x0007.010.00002763 0x00801f49.0453.01 C--- 0 scn 0x0000.389b2628
0x03 0x0002.00a.000025d5 0x00804d42.04b2.25 C--- 0 scn 0x0000.389b2811
0x04 0x0006.006.00002515 0x00800962.03c8.18 CU 0 scn 0x0000.389b3044
```

This shows some very important information on the block, especially in the ITL section shown above. This table has an *initrans* entry of four, so there are four lines, one each per the ITL. The Flag column above the flag -U- indicates that the particular ITL was used. In this case, only two of the ITLs were used, and the other two were never used. However, this is the case for this block only. By selecting block dumps from other randomly selected blocks, you could have an idea how many ITLs are actually used. Then you may decide to reduce the *initrans*.

Automatic Block Management in Oracle9i

In Oracle9i, the process of space management inside a block is somewhat changed due to the introduction of the Automatic Block Management (ABM) feature, also known as Automatic Segment Space Management (ASSM). The option is specified at the tablespace level in the storage parameter as SEGMENT SPACE MANAGEMENT AUTO. For instance, the tablespace TS1 can be created as

```
CREATE TABESPACE TS1
DATAFILE '...'
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
```

The last line of this code does the magic. In the ABM mode, Oracle maintains a bitmap for each segment with the information on the block. A bitmap is a data structure with a bit representing each block. When a block becomes available for INSERT, simply setting the corresponding bit in the bitmap rather than using freelists makes the information available. So, what does this have to do with ITL waits? The very cause of ITL waits is not freespace management, but the unavailability of a slot in ITL waits. So you still have to look for

ITL waits and correct them using *initrans* and *maxtrans*. In fact, the problem may become exacerbated because the block becomes quite packed following an efficient space management system, and that may lead to lack of space for ITL growth. You can prevent this by keeping a large *initrans* for the segment.

Conclusion

Proper setting of *initrans* and *maxtrans* and packing of the blocks is vital to avoid ITL waits in Oracle. It's interesting to note that locking doesn't cause waits, but rather, the mechanism for locking as well as and poor planning. However, the good news is that reorganizing the table and adding more slots to the Interested Transaction List can easily fix this situation.

Automated Space Cleanup in Oracle

CHAPTER 9

Automated Space Cleanup in Oracle

This article proposes a PL/SQL package that can let you quickly and easily implement automated database-wide cleanup. This can be useful for environments where large numbers of objects are processed, generating residual segments: development databases, data-warehouses data-transforms, batch-based environments, etc. This PL/SQL package is self-tuning, portable, and (almost) platform and version independent.

This solution has been tested on Unix (HP 10.7/11, Sun Solaris 7/8, AIX 4, Linux 2) and Windows servers (NT4, 2000), on Oracle 7.3.x, 8.0.x, 8.1.x. It should also work with Oracle 9i. It requires some knowledge of UNIX shell scripts, SQLPlus scripts, and PL/SQL. However, full scripts are provided and minimal knowledge should be enough to start.

Stray Temporary Segments

Part of maintaining quick response times for the database is making sure that tablespaces are free of stray temporary segments, as well as the rollback segments (RBS's) are optimally sized. Temporary segments in TEMPORARY tablespaces exist for the lifetime of the instance in order to reduce the number of recursive SQL on the data dictionary. The tablespace can also inappropriately fill up and run out of space. The new (Oracle 8.1.5) locally managed tablespaces were designed to reduce this problem. However, the majority of Oracle installations still use classical dictionary managed tablespaces.

Some events (e.g., index create or rebuild, table move or create as select, including during data-loads/imports, sorts generated by complex queries) can create a lot of temporary segments. For a discussion of temporary segments behavior during a table move / index rebuild see my article *Automated Table/Index Reorganization In Oracle8i*. If the reorg action fails or is cancelled, a lot of residual temporary segments may remain in both USER and TEMP tablespaces. Also, when a segment is dropped, initially, it is changed to a temporary segment, to allow for possible rollback in case of failure. This can also generate stray temp segments.

Important changes can also create some tablespace fragmentation, with the presence of many adjacent free space blocks. Tablespace fragmentation is not an issue in itself, but keeping track of all the extents can become a very expensive operation. Therefore the need for free space merge (coalesce) procedures. Normally, for Oracle7 and newer, the SMON process will attempt to clean up the tablespaces of these residues. For that to happen, the tablespace must have DEFAULT STORAGE (PCTINCREASE > 0).

Every five minutes SMON will scan the free extent table (*sys.fet$*) for adjacent free extents to be coalesced into single extents and will coalesce five extents at a time. Unless SMON is posted by a foreground process, it will only wake up every 25 cycles, that is two hours and five minutes, and will scan the segment table (*sys.seg$*) for temporary segments to be cleaned. Even then it will only clean up to eight temporary segments at a time, and only if it can get the required locks within five seconds. So temporary segment cleanup can appear to take a long time, hours, even days. Also, it appears that SMON has to examine all the free space extents, before actually starting the work on the first ones, which can lead to long delays in space management procedures, like creating or dropping new objects, or growing existing objects.

To do space management, a process needs the single lock ST (space transaction); if another process has got it, SMON will have to try again later. Foreground processing has priority over housekeeping, so SMON will give up the ST lock very easily. This means that a busy system which does a lot of disk-based sorting or creation of transient objects could see how SMON keeps giving up the lock before doing much work. That is why sometimes, because of the large number of extents allocated, is appears that SMON is not working or doing cleanup. Some people may sometimes use the term "missed SMON post" for this situation. The cleanup itself can also cause very high CPU consumptions.

Evaluating the numbers of temporary segments - by using the OEM Tablespace Manager, or with a query:

```
select tablespace_name, segment_name, segment_type, sum(bytes),
count(extent_id)
from dba_extents
where segment_type = 'TEMPORARY'
group by tablespace_name, segment_name, segment_type;

TABLESPACE_NAME   SEGMENT_NAME   SEGMENT_TYPE   SUM(BYTES)
COUNT(EXTENT_ID)
---------------- ------------- ------------- ---------- -----------
----
IDX01            30.53431       TEMPORARY        91996160
3
TEMP             3.19262        TEMPORARY        69058560
843
```

Manual Cleanup of Temporary Segments

SMON performs temporary segment cleanup when it is posted explicitly by another foreground process. You can make use of this fact to force SMON to clean up temporary segments more promptly. SMON is posted whenever a space transaction fails. So you can trigger temporary segment clean up by using several methods:

- in *svmgrl* issue the command 'oradebug wakeup {PID}' where {PID} is the Oracle process ID of SMON - here is a script by Steve Adams (on www.ixora.com.au) which does just that – *post_smon.sql* (http://www.ixora.com.au/scripts/sql/post_smon.sql).
- create a table and abort before completion, thus generating a signal for SMON to wake up
- create a table that will fail rapidly (e.g. use STORAGE (INITIAL 32 K NEXT 20000 M), again posting SMON
- create a small table with a primary key, then drop the key, which becomes a temp segment posting SMON to clear
- a normal shutdown will force SMON to complete clean up of temp segments, but does not coalesce
- in a PERMANENT tablespace - PCTINCREASE 0 disables SMON activity - PCTINCREASE > 0 enables it
- in a proper TEMPORARY tablespace or in a PERMANENT tablespace with PCTINCREASE > 0 the usual method is to issue an 'ALTER TABLESPACE *tablespace_name* DEFAULT STORAGE (MINEXTENTS 1);' - you can use any storage clause, and you do not have to change the current setting - this will drop temp segments, but will not coalesce the free space. 'ALTER

TABLESPACE *tablespace_name* COALESCE;' will do that for you

Starting with Oracle7 a number of events in your INIT.ORA file can be used to stop space management recursive calls - they are recommended to be used only with Oracle Support's help.

- 10061 - prevents SMON from handling temp -- e.g., event="10061 trace name context forever, level 10"
- 10269 - prevents SMON from coalescing
- 10268 - prevents forward coalescing
- 10901 - prevents extent trimming

Starting with Oracle8 you can also use an event to force the cleanup of temp segments that are not currently locked:

- *drop_segments* - set at session level, where TS#+1 is the tablespace number as in the query (select name, ts# from ts$;) and adding 1 - if the value is 2147483647, then all tablespaces are cleared
  - alter session set events 'immediate trace name *drop_segments* level TS#+1';

Similarly, starting with Oracle7 you can also use an event to force the coalesce of free space segments:

- COALESCE - set at session level, where TS# is the tablespace number as in the same query as above
  - alter session set events 'immediate trace name COALESCE level TS#';

Recommendations Regarding Temporary Segments

The classical recommendations are:

- have all user tables/indexes with DEFAULT STORAGE (PCTINCREASE 0), to obtain equally sized extents
- have all user tablespaces with DEFAULT STORAGE (PCTINCREASE 1), to allow SMON to work correctly
- TEMP and RBS tablespaces have DEFAULT STORAGE (PCTINCREASE 0), leaving the DBA to deal with this
- the default TEMP tablespace of all users (excepting SYS), should be changed from SYSTEM to something else
- if SMON is busy cleaning up TEMP segments containing a large number of extents and cannot service 'sort segment requests' from other sessions, pointing the users at a PERMANENT tablespace as their temporary tablespace can alleviate the problem and help keep the system running until SMON is free again

However, today most authors would recommend a few changes from the above:

- have all user tablespaces with DEFAULT STORAGE (PCTINCREASE 0), to stop SMON from using resources
- run a job to do cleanup once a day (e.g. at 7 a.m. before, business starts)

Locking

- to find out what users do, what they run, what resources they consume, use the script *id_sql.sql* (http://www.dbazine.com/code/id_sql.sql.txt).
- to examine locks the easiest thing is to use the Oracle Enterprise tools (Lock Manager, Top Session, etc.)

Problems with Rollback Segments

Along with increased TEMP activity, often there is increased RBS activity as well. This is not really within the scope of the present paper, but I'll still mention that RBS's make for a complex topic, and I have chosen not to attempt automatic repairs to more complex RBS problems like lost or collecting transactions, etc. This can be another paper.

Recommendations Regarding Rollback Segments

The usual recommendations are:

- the sizing and on-line/off-line switching of RBS's should be done manually
- today's authors tend to get away from allocating specific large RBS's to large transactions, advocating equally sized rollback segments, tuned to handle any transaction
- the optimal resizing/shrinking should be set to happen automatically
- an RBS should be probably sized to 10% of the largest table size, with 10 - 20 equally sized segments (*minextents* 10). *optimal* is often set to *minextents* * *minimum extent*. *pctincrease* is always 0.

Automated Space Cleanup

Our strategy can be a combination of cron jobs and a PL/SQL package (*pkg_cleansys*). I have been running the package for a few years with no problems. The execution times are very small, seconds for one object, most of the times. The package runs automatically every morning, at 7 am, for 5-10 minutes, and then the log is emailed to the DBA. There is a lot of output displayed if only run from SQL*Plus, for troubleshooting and detailed logging purposes.

Prerequisites

- you should have some system privileges (see the beginning of the *install_cleansys.sql* script. Available at http://www.dbazine.com/code/install_cleansys.sql.txt
- set *utl_file_dir* = * (or at least c:\temp, or /tmp, etc.) in *init.ora*, in order to allow log files to be created
- set *job_queue_processes* = 2 (or higher) in *init.ora*, in order to allow *dbms_job* scheduling to work

There are no associated tables. Practically, there is no hit on redo logs, TEMP space, and RBS's. Memory and CPU consumption are negligible, unless there are some serious problems, in which case you should really run the package.

Overview of the Package

The Automated Cleanup package (*pkg_cleansys*), if set so, via its parameters, attempts to:

- truncate *sys.aud$* table
- clean up temporary segments in all tablespaces

- reset *pctincrease* to 1 for tablespaces to wake up SMON
- coalesce free space in all tablespaces
- shrink all rollback segments to the optimal size
- turn autoextend off for all files
- set *pctincrease* 0 for tables and indexes, if any with *pctincrease* > 0
- grant unlimited quota to object owners
- set *pctincrease* 0 for tablespaces for the SAFE algorithm (as per Oracle white paper 711)
- set *pctincrease* 1 for tablespaces if the SAFE algorithm is not used
- rebuild unusable indexes, if any
- detect some generic unavailability conditions
- it does NOT handle locked or lost transactions
- it does NOT handle locally managed tablespaces

The code (circa 1000 lines) performs a lot of error checking and decision making in support of the commands. The results of the run are written into the /tmp or c:\temp directories. Upon completion, an email message can be sent to the DBA and the process is ready to start again.

When run manually in an SQLPlus session, display procedures ensure that debugging and detailed logging are made as easy as possible - currently many of these modules are commented out to avoid crashing the package because of overloading the server output buffer - uncomment them selectively for databases with very large numbers of objects.

Although it will not account for all situations, the package does log a wide variety of errors. The DBA will treat errors manually as the automated system will only try to re-run a session in case of failure. Some errors, like "failed because of resource busy", simply mean that a lock could not be obtained, as some other process was using the object, and can be ignored, as it will probably succeed on the next run. A number of conditions and options are also available to be enabled or disabled in the package body. If the package is run automatically with 'DBMS_JOB', we get only the *summary output*, (http://www.dbazine.com/code/DB1-CLEANsysPKG.log.txt) which can include captured error messages.

Setup

The package is installed into the default Oracle schema 'MHSYS', which I use to host my automation packages. IT CAN BE INSTALLED, AS IS, FOR UNIX AND NT BASED SERVERS. It is a pretty comprehensive piece of software, which is compatible with Oracle 7.3.x or higher, on both UNIX and NT, and includes routines to detect the current OS, Oracle version and SID.

The code is amply commented. Run the *install_cleansys.sql* script as user 'SYSTEM' from SQLPlus. Before installing, read the top of the package body, just in case you need to make some modifications. This section can also be used for tuning later, by changing the values of a number of constants. Make sure the script does not drop the existing schema 'MHSYS' if already installed. The defaults will cover most situations and, most likely, nothing will need to be changed. Sessions can vary between 1 - 10 minutes. Have the logs emailed to you or, at least, examine them manually.

You can use scripts to schedule or run the package, and to email the logs, similar to the ones described in my article *Automated Cost Based Optimizer* (Oracle Magazine Online - Sept 2000). For a list of Frequently Asked Questions and tips on running my packages, you can visit www.hordila.com/mhwork.htm.

Using Oracle TEMP Files

CHAPTER 10

Temporarily Yours: Tempfiles

I'm sure you've spent countless evenings discussing deep philosophical questions like, "If a tablespace that no one knows about drops in the woods, do we need to recover?" With Version 8.1, Oracle has finally answered this question with a solid, "No."

Every user has a temporary tablespace that's behind the scenes to be used only for the duration of statements. You should not be able to put any permanent object in this tablespace. All users usually share it. Now the maintenance of this temp tablespace just got a whole lot easier.

Don't Wait to Create

Before 8.1, every time you created a tablespace, you needed to wait while all the blocks were created in the file. This could take a fairly long time, and seemed to be in proportion to how long it had been since you last slept. With Oracle 8.1, the new temp file is created in seconds.

```
create temporary tablespace TEMP01
tempfile '/oracledb/oradata/lcav/temp0101.dbf' size 1024M
extent management local uniform size 1M;
```

Notice that we're creating a temporary tablespace on a temporary file. This will take the 1G on the mount point, but does not spend the time to write the block headers through the complete file.

Don't Backup

Since these temporary tablespaces don't hold any objects and are created in seconds, there's no reason to back these up. So in your hot backup script, you can exclude the tablespaces that are built on temporary files.

```
cursor c1 is
select tablespace_name from sys.dba_tablespaces
minus
select tablespace_name from sys.dba_temp_files;
```

We do not spend the time copying the file and don't need to allow space for this file.

Don't Recover

Of course, since we didn't back up the file, we don't recover it. If you execute:

```
Alter database backup controlfile to trace;
```

you'll see the following after the create statement for the controlfile:

```
# Commands to add tempfiles to temporary tablespaces.
ALTER TABLESPACE TEMP01 ADD TEMPFILE
'/oracledb/oradata/lcav/temp0101.dbf' REUSE;
# End of tempfile additions.
```

The tablespace exists, but there's no file under it until you add it. Oracle doesn't need to do anything to this file during recover, so no time is spent on it.

Don't Copy for Standby

A corollary for these points is that the file isn't needed for the standby database. You will need it if you're going to use it as a read-only instance in that case, simply add the tempfile to the tablespace as with the recover process previously noted.

Don't Add Overhead

Because it is temporary and managed locally, we're not adding any overhead to the system tablespace. There's no logging being done, and a coalesce is never performed. Since all of the extents are the same size, there's never a need to coalesce the free extents.

This does mean that we're not going to see any mention of this tablespace or file in *dba_extents*, *dba_segments*, or *dba_free_space*. Oracle has given us new views to look at for these temp files.

I use *v$temp_extent_pool*, *v$tempstat*, and *v$temp_space_header* the most. You'll still use *v$sort_usage* to see who's doing what in the temporary tablespace (from my Dbazine article, *Who Took All the Temp?*):

```
set pagesize 10000
set linesize 133
column tablespace format a15 heading 'Tablespace Name'
column segfile# format 9,999 heading 'File|ID'
column segblk# format 999,999,999 heading 'Block|ID'
column blocks format 999,999,999 heading 'Blocks'
column username format a15
select b.tablespace,b.segfile#,b.segblk#,b.blocks
,a.sid,a.serial#,a.username,a.osuser,a.status
from v$session a
,v$sort_usage b
where a.saddr = b.session_addr
order by b.tablespace,b.segfile#,b.segblk#,b.blocks;
```

Give It a Try

So add a tempfile and set it as your temporary tablespace. You can see the usage from the v$ tables. Now you'll have much more time to get back to the important questions like, "How many DBAs can dance on the head of a pin?"

Monitoring TEMP Space Usage

CHAPTER 11

Who Took All the TEMP?

We have a user that submitted a query with a three-way join with an 'order by.' No problem. This happens every day. The difference here is only two of the tables were joined so we had a Cartesian product runaway. Our temp tablespace was set to auto-extend, so the query did complete. Of course, it wasn't the result set that the user wanted.

Unfortunately, we didn't even notice until we did file backups that night, at which point it no longer all fit. So after careful thought and consideration, it occurred to us that we really didn't have a good grasp of what was happening in the temporary tablespaces. And that notion compelled us to investigate and find out answers to the real question: "Who took all the TEMP?"

Where Are My TEMP Tablespaces?

First we needed to find the tablespaces that could be affected by this. It is not defined by the contents column in the *dba_tablespaces* table, but better defined by the following query:

```
Select distinct temporary_tablespace from dba_users;
```

There can be multiple tablespaces fulfilling this function but there is usually only one that the end users are assigned to access. So this is the one we will investigate.

Show Me the Objects

Now that I have the tablespace name I can query *dba_extents* to see the allocated extents;

```
set pagesize 10000
set linesize 133
column tablespace_name format a10 heading 'Tablespace|Name'
column file_id format 9,999 heading 'File|ID'
column block_id format 999,999,999 heading 'Block|ID'
column blocks format 999,999,999 heading 'Blocks'
column segment_name format a15 heading 'Segment Name'
column segment_type format a15 heading 'Segment Type'
break on tablespace_name skip 1 on file_id skip 1
select tablespace_name, file_id, segment_name
,segment_type, block_id, blocks
from dba_extents
where tablespace_name = 'TEMP'
order by file_id, block_id;
```

The first rows of my output look like the following:

TABLESPACE NAME	FILE ID	SEGMENT NAME	SEGMENT TYPE	BLOCK ID	BLOCKS
TEMP	20	20.1042	TEMPORARY	1,042	260
		20.1042	TEMPORARY	1,302	260
		20.1042	TEMPORARY	6,782	260
		20.1042	TEMPORARY	7,042	265
		20.1042	TEMPORARY	7,307	265
		20.1042	TEMPORARY	7,572	270

Who Are the Users?

This is interesting output but the question still remains: who is using those objects? As of Oracle 8.0 we have a new view to easily see this.

```
column tablespace format a10 heading 'Tablespace|Name'
column segfile# format 9,999 heading 'File|ID'
column segblk# format 999,999,999 heading 'Block|ID'
column blocks format 999,999,999 heading 'Blocks'
break on tablespace on segfile#
select b.tablespace,b.segfile#,b.segblk#,b.blocks
from v$sort_usage b
order by b.tablespace,b.segfile#,b.segblk#;
```

This gives the following:

TABLESPACE NAME	FILE ID	BLOCK ID	BLOCKS
TEMP	20	1,042	6,979

This new view gives me basically the same information as *dba_extents*, except it only shows the starting block number, not the block number for each extent. So where I had 78 rows above, here I have 7 rows.

Let's add in the session information so we can see who the user is.

```
column username format a10
select a.tablespace,a.segfile#,a.segblk#,a.blocks
,b.sid,b.serial#,b.username,b.osuser,b.status
from v$sort_usage a
,v$session b
where a.session_addr = b.saddr
order by a.tablespace,a.segfile#,a.segblk#;
```

Output:

TABLESPACE NAME	FILE ID	BLOCK ID	SID	SERIAL #	USER NAME	OSUSER	STATUS
TEMP	20	1,042	10,139	426	4517	OPS $ORACLE	ACTIVE

With these results, we now have the sid and serial number, so we can kill the session. The hitch here is that the user can simply resubmit it. What we can then do is refine our question to be: "What statement is causing the explosion in TEMP?" To get this answer we just join with *v$sqltext*:

```
Break on tablespace on sid on serial# on blocks
select a.tablespace, b.sid, b.serial#, a.blocks
,c.sql_text
from v$sort_usage a
,v$session b
,v$sqltext c
where a.session_addr = b.saddr
and b.sql_address = c.address
order by a.tablespace,b.sid,b.serial#,c.address, c.piece;
```

Output:

TABLESPACE NAME	SID	SERIAL #	BLOCKS	SQL_TEXT
TEMP	426	4517	21,274	select * from appointment order by start_time

As you can see, we finally the answer to our question. This view is changing constantly but we are really looking for the query that seems to be stuck in there. Statements that need help will stay in the result set, and you will see the numbers of blocks continue to increase.

A Happy Ending

Now I can capture the code before killing the runaway session. This way I can talk the user through some changes before they submit it again.

We can find the same query by looking at all active queries. But this additional view gives me more documentation when going to the user and also removes out any doubt that I have the right query without killing and running again.

CHAPTER 12

Oracle9i Self-Management Features

Oracle9i Self-Management Features: The Early Winners

Introduction

BMC Software sells a number of tools to help manage Oracle dataservers. As each new version of Oracle moves into production we need to form a view on the additional features that may be required in our products to help our customers fully exploit their Oracle instances, and also to identify those product features that will no longer merit continued development investment because Oracle has met within their software a need which had traditionally required an external solution. This paper reports some of the findings from a study of the initial production release of Oracle9i. The sole purpose of the study was to discover the extent to which Oracle had increased the ability of the dataserver to manage itself. The study took the form of structured experiments rather than observation of production use. The conclusions are therefore tentative until confirmed or disproved by some brave pioneer site, giving rise to the words "early winners" in the title of the paper.

The paper does not cover the new Oracle9i features in the areas of backup and recovery, nor does it discuss Oracle's Real Application Clusters (RAC), the replacement for Oracle Parallel Server.

No suitable clustered platform was available to the author to install and run the Real Application Cluster support before the deadline for submission of this paper, and it was not felt useful to report the 'paper study' that had been completed. As more and more sites seek to become at least more failure tolerant, if not completely non-stop, it is expected that managing multiple instance Oracle will become a major growth area and the author hopes to extend this paper to cover the topic.

Backup and recovery are not discussed for a number of reasons, not the least being that the paper is based solely on experimental use of the software and the author believes that testing in a near production environment is essential to valid commentary on data security issues. It is also becoming clear that as the number of features in the Oracle dataserver continues to increase with every release, even server specialists are starting to have some difficulty keeping fully up to date in every area. The author's key focus in undertaking the technical studies on which this paper is based was to look at new manageability features primarily from a performance and availability standpoint.

Test Environment

All of the testing for this paper was performed on the author's laptop, a Compaq Armada M700 with 448 Mb of memory and a single 20 Gb hard disk partitioned into a FAT device for the operating system and an NTFS device for both the Oracle installation and the database files. The single processor is a Pentium III with a reputed clock speed of 833MHz; it certainly executed queries from the Oracle9i cache at impressive speed.

The machine was running Microsoft Windows/2000 Professional with Service Pack 2, and Oracle9i Enterprise Edition release 9.0.1.0.1 with Oracle's pre-packaged "general purpose" database although this was customized in a number of ways. Most importantly the INDX tablespace was removed, the USERS tablespace was expanded to 2 Gb, and limits were set on datafile autoextension in each tablespace. It was noted with some disappointment that the default databases all came with a database block size of 4096 whereas the author would have preferred to use 8192. This presented an opportunity to test operation with multiple block sizes but these tests were not completed in time to be included in this paper. It is hoped to make a brief comment on the results during the conference presentation.

Self-Management

Goals

We're all busy, and Database Administrators (DBAs) are often busier than most. It makes huge sense that if the dataserver is completely capable of taking whatever action is required, then it should do so without waiting for a DBA to come along and give it permission. On the other hand, customers are not impressed when a dumb machine keeps making things worse by applying inadequate management rules, especially if these are enabled by default as always seems to be the case in some personal productivity software.

At is simplest, therefore, we can establish two goals for self-management features:

- If the server can successfully manage the situation, then it should do so.
- If success is in doubt, then management should be left to "someone you can fire."

Examples

In the world at large, automobiles contain a number of features that are essentially self-managing including anti-lock brakes and automatic transmission. In general, a skilled enough driver can get better performance from the vehicle without these systems, but most of us have better things to do and opt for the increased safety and ease of use of the self-managed systems rather than insisting on exercising the maximum level of control.

Self-management features are not new in Oracle. Version 6 was the first version with a region called the shared pool, but it also had a discrete data dictionary cache divided up into separate sections for tables, objects, indexes, columns, and so on. Each of these regions had to be sized using its own *init.ora* parameter and the penalty for making any one of them too small was a significant increase in parse times. Pretty soon Oracle worked out that most customers were failing to size these regions accurately and decided that the dataserver should size the cache dynamically from the memory allocated to the shared pool. In the case of the row cache (Oracle's slightly bizarre internal name for the data dictionary cache) the user no longer has to do anything at all. It sizes itself. (If you are interested and want to see how many failure-prone decisions this is saving you, take a look at *v$rowcache*.)

When it comes to the log checkpoint interval we have a trade off. A longer gap between checkpoints means a higher maximum possible transaction throughput, but also increases the average and maximum instance recovery times. In Version 6, the DBA was expected to adjust checkpointing by reference to the number of redo log blocks that had to be written before a checkpoint took place; not surprisingly, many DBAs found it difficult to relate this integer to the mean or maximum recovery times. Subsequent releases have made the specification progressively easier, and in Oracle9i, the DBA simply specifies the desired mean time to recover (in seconds). The administrator specifies the goal, and the server does whatever it believes needs to be done to meet that goal.

Instance Parameter Management

Each of the recent versions of Oracle has had between 100 and 300 run-time parameters that can be specified at startup and tell the instance "whether it is a prune Danish or a fruit cake". My copy of Oracle9i has 250 of these parameters that I am allowed to set (even if I am discouraged from doing so) and an additional 432 parameters whose names start with an underscore, which I am not supposed to change unless told to do so by Oracle Support. *_allow_read_only_corruption* = true is just one example.

Over the years it has become less and less acceptable to stop database servers to adjust instance parameters, and Oracle has made many of the parameters 'dynamic' meaning that a suitably authenticated user can modify the value of the while the database is running. Until Oracle9i these dynamic changes were lost when the instance was stopped and restarted because at restart the parameters were read from a text file invariably referred to by its original default name *init.ora* although it should really be known as a *pfile*.

Oracle9i can use an alternative type of parameter file called an *spfile*, the key difference being that the new file type is maintained from within Oracle. Thus the SQL statement alter system set *db_cache_advice* = on scope = BOTH; enables the cache advice feature within the current instance, and also specifies that it will be enabled on subsequent restarts. The scope argument may have the values MEMORY, SPFILE, or BOTH.

Better still, the manuals tell us that even more of the parameters are now dynamic, including those that set the size of the shared pool and the various buffer pools (in order to support multiple database block sizes, Oracle9i allows one buffer pool per block size plus default, keep, and recycle pools for the default block size). The manuals also state that even the parameters that cannot be changed dynamically can be altered using alter system set commands with scope -= spfile. Unfortunately the reality is less encouraging. The total amount of memory allocated to the SGA cannot be changed and so in order to increase one allocation, another allocation must be first reduced. Also a number of important memory regions (including the java pool size and redo log buffer) cannot currently be resized.

Worse, there are severe implementation problems. It does not appear to be possible to change static parameters as advertised in the manuals as can be seen from the following transcript from SQL*Plus:

```
alter system set java_pool_size  = 30M scope = SPFILE
                 *
ERROR at line 1:
ORA-02095: specified initialization parameter cannot be modified
```

Finally (at least under Windows/2000) any attempt to resize a buffer pool with the *db_cache_advice* feature enabled causes the instance to crash, leaving an ORA-00600 in the alert log. The cache advice feature, discussed below, is strongly recommended, but however attractive the idea of the spfile seems from the documentation, it clearly needs to be deployed with extreme care (if at all) in the current release.

Self-Tuning Memory Management

Oracle9i has two major features that can be used to help overcome the memory conundrum discussed below. The first is completely automatic and simply strives to keep memory usage below a specified level. The second gathers and presents data on the likely effect of changing the size of a buffer cache, allowing the DBA to determine whether or not cache sizes are appropriately set and facilitating tradeoffs between the buffer pools and the shared pool or PGA space.

Memory Made Simple

Effective use of memory is key to Oracle performance. Memory access is several orders of magnitude faster than disk access, and most high-performance Oracle applications are inherently more limited by their I/O capacity than by their CPU capacity. Many technical authors and DBA's conclude from this that the more they keep in memory the faster their applications will run, but unfortunately life is not quite that simple.

Firstly there is little point in keeping something in memory if it is not going to be used again before it eventually gets overwritten. Secondly Oracle is often (usually?) run using 'dedicated servers' in which case each Oracle session has its own operating system thread or process with a private and potentially large memory region called the PGA (Program Global Area). As the number of sessions increases so does the amount of PGA space required and eventually the operating system has to resort to paging in order to meet the demand. This can, of course, happen with only one Oracle session if either the buffer cache or the shared pool is set pathologically large, or the platform simply does not have enough memory to run the Oracle version effectively.

Once demand paging becomes a significant factor, much of the data that appears to be in memory has to be managed on disk by the operating system and this defeats the original purpose of allocating large memory work areas. The goal, therefore, is to keep the memory areas at a size that minimizes I/O by retaining frequently used items without causing excessive paging. The difficulties are that it is not usually clear how a change in cache size will affect I/O, and the number of sessions may vary over time making it difficult to decide how much memory the DBA can afford to allocate per session. The parameter *sort_area_size* is a particular challenge in this regard because large sort work areas can dramatically improve the performance of certain operations (typically index creation, derivation of object statistics and reporting) but they are normally unnecessary and costly for most users.

PGA Aggregate Target

Traditionally Oracle has supported has series of instance parameters such as *bitmap_merge_area_size*, *open_cursors*, *open_links*, and *sort_work_area* that together determine the size of the PGA. However whereas the value of a parameter such as *open_cursors* makes only a small difference to the overall memory charge, changing *sort_work_area* from 65536 to 1048576 could under the worst circumstances alter the total memory charge by over 4 Gb in a 5,000 user system.

The new instance parameter *pga_aggregate_target* does pretty much what it name suggests - it instructs Oracle to try to keep the total space used by PGA's within the instance below the target value. It cannot guarantee to achieve this 100% of the time for reasons that are covered in the Oracle documentation, but it does offer the DBA the opportunity to make the size of bitmap merge and sort areas vary with the number of simultaneous users that need to use such an area. Unfortunately it was not possible to test either the performance or the overhead of this feature in the short time between the Oracle9i production software becoming available to the author and the deadline for the submission of conference paper. It looks like a potentially valuable and completely automatic feature that will really help sites where the number of sessions is subject to great variation. However it should be noted that current best practice in the architecture of 3-tier applications does not give rise to such variations, preferring to use a constant size pool of server sessions. To give the final word on this topic to the Oracle9i Performance Guide and Reference "Oracle Corporation strongly recommends switching to the automatic memory management mode, because it is easier to manage and often outperforms a manually-tuned system."

Cache Advice

As already hinted, part of the mythology of Oracle performance tuning is the idea that the larger the buffer pool, the better performance Oracle will give. Before Oracle Version 8.0 large buffer pools dramatically increased the CPU power needed for a given workload, but this is fortunately no longer the case. However as already discussed it is still desirable to keep the buffer pools only just large enough in order to make memory available for other regions, in particular the shared pool and the session work areas (the PGA or, in a shared server environment, the UGA).

For many years Oracle had a pair of linked features, one of which that tracked the effectiveness of the buffer pool and the other that tracked the predicted effectiveness of proposed cache extensions. Unfortunately not only were these features awkward to interpret, they incurred an unacceptable CPU load. These discredited features have been replaced in Oracle9i by a so-called cache advice mechanism; this does not quite live up to the promise of its name, but it is nonetheless worth exploring.

There is a new instance parameter *db_cache_advice* that may be set to on, ready or off. The default is off but the documentation suggests that the trick is to set it ready because setting it from off to on requires a memory allocation that may fail. Once the feature is enabled the performance of each buffer pool is tracked to determine how effective it is being at reducing physical reads; the results are available from the virtual view *v$db_cache_advice*. The Oracle9i Reference contains the warning that "CPU and memory overheads are incurred" but in a quite punitive test the author found that the CPU overhead was consistently less than 10%. This seems a reasonable cost to incur from time to time for the benefit of being able to correctly size the buffer caches. The view *v$db_cache_advice* shows, for each buffer cache, a series of statistics on the estimated number of physical reads that would have taken place if the buffer cache had been that size over the period since either startup or when the feature was enabled. Sample output from a reporting script is shown below.

```
Cache        Cache   Total Est Read   Est Phys
Name         in Mb Buffers   Factor      Reads
---------- ------- ------- -------- ----------
DEFAULT       6.14   1,572     1.19     18,560
DEFAULT      12.28   3,144     1.11     17,418
DEFAULT      18.42   4,716     1.01     15,783
DEFAULT      24.56   6,288     1.00     15,658
DEFAULT      30.70   7,860     1.00     15,658
DEFAULT      36.84   9,432     1.00     15,638
DEFAULT      42.98  11,004     1.00     15,596
DEFAULT      49.13  12,576     1.00     15,591
DEFAULT      55.27  14,148     1.00     15,589
DEFAULT      61.41  15,720     1.00     15,587
```

The default buffer cache was 20 Mb in this example, but when the instance was started it had been 32 Mb and was reduced using alter system before running with cache advice enabled. Although the author has not found documentation to confirm this, from observation the cache advice feature monitors the predicted effect of buffer pools from about 20% to 200% of the size of the pool at startup in steps of 10% of the original size. The query that produced the example intentionally removed every second step in order to present a shorter table.

Although the view does not actually project any advice as such, two conclusions can quickly be drawn from the output in the test case shown. If only 6 Mb of buffer space had been allocated then physical reads would have increased by about 20%, whereas allocating more than 20 Mb of buffer space would have hardly reduced physical reads at all. The estimate is that the addition of the final 12 Mb would have reduced physical reads by just 4, or about 0.025%. Had this sampling interval been reasonably long and taken against the instance under typical load then we would have proof that 20 Mb was the correct buffer pool size for the service being delivered.

Data that demonstrates the ineffectiveness of enlarging the buffer pools is invariably hotly disputed by part of the Oracle community, but the mathematics has been known for many years. Hopefully this new mechanism will gain both credibility and use, and help Oracle DBA's to reach better compromises in memory allocation.

Automatic Undo Management

Background

Other than the dreaded ORA-00600 the Oracle error that seem to strike the greatest fear into the hearts of DBA's is ORA-01555, the "snapshot too old" error (though to be fair ORA-03113 is another one that you really do not want to be faced with). Snapshot too old means simply that the read consistency mechanism has tried to reach too far back in time.

There are two reasons why a read consistency operation may need to be performed. Firstly a transaction may try to look at data that is subject to another user's uncommitted changes. If Oracle is working properly then this read consistent operation is guaranteed to succeed because Oracle is required to be able to rollback or undo a transaction at any time until it is committed.

Each query has a read consistency point, and all of the data returned by the query must be the committed state of the data as of that point in time. Normally the read consistency point is established by executing the query, but it can also be established for a whole series of queries by starting a "read only transaction". The ability to reconstitute the data as it appeared before a committed change is not guaranteed; in most applications it works without problems for most of the time but from time to time users experience the "snapshot too old" problem.

Rollback Segments

Each time Oracle makes a change to schema data it records the information required to undo that change in a special type of database area called a rollback segment. This information is always kept at least until the transaction making the change has committed, but as soon as the transaction is complete its rollback or undo data can be overwritten. How soon this happens depends on how much undo space is available and how quickly current and future transactions create new undo records. Within a few seconds, or minutes, or hours the undo information will be overwritten or, in some cases, simply discarded.

Since the introduction of Oracle Version 6 in 1988 the allocation of rollback segment space has been a major concern for Oracle DBA's who have had to decide both how many rollback segments an instance should have and how large each one should be. Resolving this issue has typically required a number of compromises that are outside the scope of this paper.

The Oracle9i Solution

Oracle9i supports the traditional rollback segment management features that have evolved over the past 13 years, but also introduces Automatic Undo Management. In this mode the DBA only has to create an "undo tablespace", tell Oracle to use this tablespace, and specify for how many seconds each undo record must be retained. The records will, of course, be kept for longer if the transaction that creates them does not commit within the time interval.

In Oracle9i the following three instance parameters will guarantee that all undo entries will remain available for 15 minutes:

```
undo_management = AUTO
undo_retention  = 900      # seconds
undo_tablespace = UNDOTBS
```

However a potentially unwanted side effect is that the Oracle server will not retain the data for much longer than the time specified even if the instance is running with a relatively light updating load i.e. even if there is no great demand to write new undo information. This contrasts markedly with traditional rollback segment management, where under light updating loads undo entries could (and would) remain available for several hours to generate read consistent data sometimes required by long running reports. Fortunately the instance parameter *undo_retention* can be altered dynamically using alter system set and this may become necessary at sites which have long report runs take place and cannot completely prevent update from occurring while these reports are running.

Automatic undo management looks like a winner despite the likelihood that many sites will find it necessary, or at the very least desirable, to alter the retention period dynamically. The parameter is specified in terms of time rather than the present specification in blocks, which requires the DBA to assess how much undo his instance generates per second and to hope that this remains approximately constant.

Database Resource Manager

Database resource management is present in Oracle8i, though as far as the author can discover the customer base has not used it extensively. Changes to the way in which Oracle manages user session processes (threads under Windows/NT and Windows/2000) have resulted in a number of changes under the covers but the basic functionality remains the same, and contrasts markedly with the user profiles feature.

Profiles, when enabled, set hard limits on the amount of resource that a session may use and may also be used to limit to number of sessions that a given user ID may start in parallel. Both CPU and logical reads (database block visits) can be rationed at either or both session level and call level. Thus a profile might limit a particular user to never exceeding 60 seconds CPU time in any call to the database. Profiles also have a role in password management, but this is outside the scope of this paper. The major problem with hard limits on block visits and CPU consumption is that such resource usage does little or no harm if other users of the server are getting the resource that they require.

In its simplest usage the Database Resource Manager seeks to control the share of the CPU resource allocated to specific groups of sessions rather than the total amount of resource consumed by those sessions, and to intervene only when some group is in danger of not receiving their allocated percentage. However there are a number of other features including rationing the number of sessions that a specific group may start and applying execution time limits to database calls. These time limits are significantly different from the limits applied by profiles because the operation is aborted when Oracle predicts that it will overrun its limit rather than waiting for the limit to be exceeded.

Database Resource plans can quickly become extremely complex, with sessions being migrated from one resource group to another as they consume increasing amounts of resource. However in limited testing by the author, operating a simple resource plan appear to use remarkably little resource and it did prevent long-running queries from monopolizing the CPU. In view of the extreme problems associated with trying to prioritize Oracle users through operating system scheduling features the database resource manager looks to be a valuable feature for sites that area likely to experience CPU resource conflicts. The other side of the coin is that under heavy load many Oracle instance bottleneck on I/O resource rather than CPU, resource plans can only tackle this indirectly.

Unused Index Identification

Indexes consume disk space and lengthen backup and restore operations; in addition index maintenance during insert, update and delete operations is a major CPU and I/O overhead in many Oracle applications. There is therefore a clear incentive for the identification and removal of both unused indexes and those indexes which are either little used or make no contribution to application performance. Having spent much of the past year working in this area under Oracle8i the author was fascinated to discover that Oracle9i contained a feature specifically designed to identify unused indexes. The new syntax alter index <*index_name*> monitoring usage; creates a row for the index in the view *v$object_usage*. This view contains a column USED that is set to YES when the parser first includes the index in an execution plan.

No indication is given of how often the index is used or whether or not its use was beneficial from a performance viewpoint. More worrying, the author's initial tests indicated that the column could be set to YES even if the index had not been used. This result is so bizarre that the author is seeking independent confirmation of it and intends report further during his conference presentation. It is as yet unclear whether the feature is intended to report index usage that does not appear in the execution plan.

For sites at which parsed SQL statements remain in the shared pool for a reasonable amount of time even if only executed once (usually the case for application that make effective use of bind variables) the simple query

```
select INDEX_NAME
  from USER_INDEXES
 minus
select OBJECT_NAME
  from V$SQL_PLAN
 where OBJECT_OWNER = USER;
```

will identify indexes that have not been used by any statement currently in the shared pool. This has proved reasonably effective at detecting unused indexes though it is known to miss indexes that are used solely for constraint enforcement. *v$sql_plan* is another new and extremely welcome feature in Oracle9i. We always knew that the execution plans must be in the shared pool, and now we can report them with almost trivial SQL queries.

Oracle Managed Files

Oracle stores its data, including its log files and control files, in storage domains managed by the underlying operating system and relies on operating system services (albeit at a fairly low level) to handle data transfers between memory and disk. Although raw devices may need to be configured under some circumstances, the data is normally contained within a file system and each file is referenced by both a path and a name. Thus on my laptop the SYSTEM tablespace is stored within the single operating system file D:\ORACLE\ORADATA\DAE901\SYSTEM01.DBF. It should be no surprise on a small "server" such as my laptop that the files used to hold tablespace data for that particular database all share the same path, in this case D:\ORACLE\ORADATA\DAE901 (the database and the database instance are both called DAE901 after my initials and the Oracle version number).

Oracle Managed Files allow an administrator to specify a default path or location for database files, and this means in turn that operations that previously required the user to provide a file path and name can now be carried out with Oracle providing a default file. Thus (with suitable privilege) the two SQL*Plus commands:

```
SQL> alter system set db_create_file_dest = 'D:\TEMP';
System altered.
SQL> create tablespace XXX;
Tablespace created.
```

result in the creation of the file D:\TEMP\ORA_XXX_ZY2TFB00.DBF which will be used to hold all of the data stored in that tablespace. This file has been created with the default size of 100 Mb and with autoextend on, but both of these attributes could have been overridden in the create statement without having to specify a file name. If the tablespace were now to be dropped using the SQL*Plus command

```
SQL> drop tablespace xxx including contents;

Tablespace dropped.
```

then not only would storage segments (such as tables) in that tablespace be dropped, and the tablespace removed from the data dictionary, but Oracle would also delete the operating system file that it created during the create tablespace operation. The ability to create the required files by default is also supported in the create database command.

Such functionality may not be of immediate use to many Oracle DBA's who are well-used to allocating file space for Oracle's "container files" and who also expect to have to take decisions on where in the file system such data should be located for capacity and load balancing. However for a group trying to write install and maintenance procedures for an Oracle database that is to run on many machines whose device configuration is unknown in advance, the facility to allow Oracle to name the files means one less important error-prone step to be carried out by the installer.

Oracle Managed Files may be a minor misnomer as for all normal operating purposes Oracle "manages" the file in the same way as any other database file, but the functionality will be very helpful to a number of third party software suppliers.

Conclusions

From initial experience of the self-management features discussed in this paper, the early winners are the facility to allow Oracle to determine how much memory to allocate to individual sessions and the ability to set CPU time allocation targets for groups of users. In addition automatic undo management and Oracle managed files both look promising but are perhaps more likely to be adopted for new databases than as part of an upgrade strategy as the problems that they solve should already have been overcome in any pre-existing production application.

The ability to have Oracle report unused indexes looked attractive at first sight but was not found to be useful in practice. The spfile mechanism, which is provided to allow administrators to make persistent changes to instance parameters through the alter system command, should be extremely attractive. Unfortunately in the first production release of Oracle9i the feature has a number of failings and should be used. The associated ability to dynamically resize Oracle's caches is useful, but somewhat incomplete. It does not support all of the caches, and also it does not allow increase of the total amount of memory allocated to the caches.

CHAPTER 13

Internals of Locally-Managed Tablespaces

Locally Managed Tablespaces

Locally managed tablespaces were introduced in Oracle 8i, and have slowly been gaining popularity. However, the take-up is still a little slow. This article describes what are locally managed tablespaces, why they are good, and offers strategies for using them.

Tablespaces Past and Present

A tablespace is a logical unit of storage. It can span many data files, and contain many data segments. The available space in a tablespace is broken up into extents, and each data segment is made up of one or more extents, in which an extent is simply a contiguous section of a single data file. Typically, a data segment corresponds to a self-contained and cohesive collection of data (such as a table, or an index partition) that has some meaning to an end user.

This outline of a tablespace immediately introduces two space management issues. First, which extents belong to which segment; secondly, which extents are in use and which are available. The methods of addressing the first issue have not changed (much) in recent years, but Oracle Corp. has introduced locally managed tablespaces (LMTs) to address problems associated with the second issue.

The Past

Historically, space management in a tablespace was handled through a couple of tables, *uet$* (used extent table) and *fet$* (free extent table).

When you needed to allocate some space to a segment, Oracle would search *fet$* for an entry describing an extent of an appropriate size in the correct tablespace. In fact, there were a number of complicated little strategies that Oracle used in this search that could take some time to complete — but eventually, Oracle would delete (or modify) a row in *fet$* and insert a row into *uet$*.

Similarly, when you freed up an extent (by dropping a table, say) Oracle would delete a row from *uet$* and perform a row-insert or modification on *fet$*.

In fact, the entire process could also require some changes to be made to a row in *seg$* (the table describing data segments) and *tsq$* (the table describing space quotas that had been allowed to users). Moreover when you added an extent to a segment, a map in the segment's first block (the segment header block) had to be updated to record the fact that the extent was part of the segment.

So, all the work regarding space management for all the tablespaces in the entire database focused on two critical tables in the data dictionary (hence, dictionary managed tablespaces or DMTs.) Unless the DBA really knew what was going on and was totally in control of the system, this could cause problems. There were three main reasons why problems could appear.

First, Oracle Corp. had decided to protect all space management operations under a single space transaction enqueue ("ST" lock), rather than one lock per tablespace. So if a number of jobs were making heavy demands for space management, they could easily end up queueing up for the lock and wasting processing time.

Secondly, Oracle Corp. effectively encouraged DBAs to generate a demand for space management tasks by introducing various segment-level storage parameters (such as initial, next, pctincrease) that promised a spurious degree of precision in storage requirements but defaulted to values that guaranteed that space management would be poor.

Finally, there was enough ignorance and uncertainty in the marketplace that it was easy for junior DBAs to follow procedures that more or less guaranteed that if something could go wrong, it would go wrong. All the problems relating to DMTs can be avoided — so long as someone gives you enough time to find out how they really work (and does the modern DBA ever get the time they need?).

The Present

An LMT is responsible for its own space management. In a clean system, every file in a tablespace is sliced into equally-sized chunks (with the exception of the first 64K of the file, which is used to store a bitmap identifying which of the other chunks are currently in use). Each bit in the bitmap corresponds to a chunk in file — if a bit is set, the chunk is in use, and if a bit is clear, the chunk is free. Figure 1 shows a schematic of a newly created tablespace containing a single file, and the state of that file after some object creation and dropping has occurred.

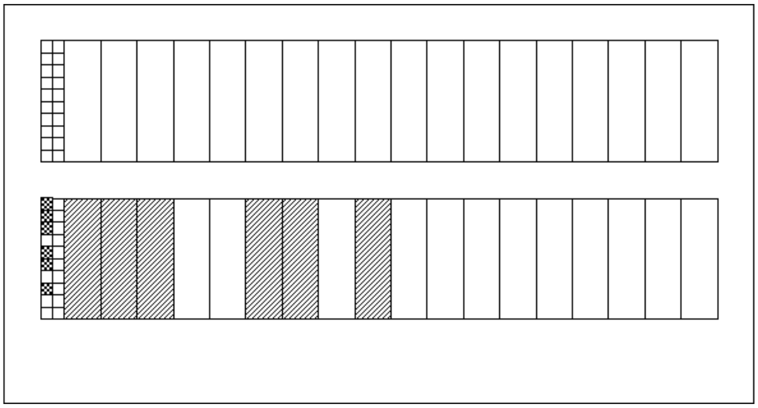


Figure 1: *A Clean LMT (upper) and a partly used LMT (lower).*

You can specify the size of the file, and the size of the chunks to be used in the file. For example, consider the script:

```
create tablespace demo_01
datafile 'c:\oracle\oradata\D9202\demo_01.dbf'
size 102464k
extent management local
uniform size 1024K
;
```

This creates a tablespace with a single file (with a very fussy size declaration), which is sliced up into exactly 100 chunks of 1024K, but has an extra 64K specified to cater for the file's bitmap. If we query the *dba_free_space* view to find out about the free space in this tablespace, we will find that Oracle reports exactly 104,857,600 bytes. When we try to allocate an extent, we will find that the extent will be exactly one chunk — under uniform size management, one chunk equals one extent.

A common problem with LMTs is that DBAs declare the file size without catering for the bitmap space; consequently they "lose" most of an extent's worth of space at the end of the file because the file is just 64K too small to create the last extent. If this happens to you, all you have to do is resize the file to add the missing 64K, and you may suddenly discover a whole extra extent appearing in *dba_free_space*.

Note: there is an autoallocate option for LMTs that can be used instead of uniform size X. This still slices the file up into uniform chunks (in this case, always at 64K), and uses one bit per chunk. However, instead of equating one chunk with one extent, Oracle will consider past history and available gaps to decide what size extent to allocate. The extent will be one of a limited set of sizes — 64K, 1MB, 8MB, 64MB, or 256MB. For relatively small, simple systems in which there isn't much information available about proper sizing requirements, this can be a minimum fuss mechanism to adopt; but in general, you should stick with uniform sizing.

So what difference do locally managed tablespaces make? Most significantly, whenever you allocate space in an LMT, Oracle does not have to search through a table to find a row that describes a suitable chunk; instead, it just scans the first few blocks of the file looking for the first free bit, and sets it. This is a much more efficient method of finding and allocating space, and has the pleasant side-effect that free space near the start of the file will be preferentially allocated — which may help to keep file sizes small, and eliminate redundant effort in rman backups.

Of course, Oracle still has to worry about the *seg$* table and the segment header block, and may still have to update the *tsq$* table, but the most labor-intensive part of the operation becomes a lot more efficient. Moreover, instead of using a single space transaction (ST) enqueue to cover the entire database, Oracle uses a new enqueue type — the TT enqueue

— and allows one TT enqueue per tablespace to reduce problems of contention due to simultaneous space transactions.

There are trade-offs. It is now much quicker and cheaper to allocate and de-allocate space, but some of the classic reports for summarizing free space or used space just got more expensive. Instead of querying a single table (*fet$* and *uet$* respectively) you now have to visit every file header to get a summary of free space, and every segment header to get a summary of used space. However, performance is not really the issue, and such reports need not be run frequently (I hope).

Where Are the Benefits?

There are always three areas in which a new feature might be of benefit: (a) strategic direction, (b) performance and (c) administrative ease. I shall address each topic in turn.

Strategically, you should be moving your systems to LMTs. Under Oracle 9.2, the database creation assistant will by default create your database with the system tablespace declared as an LMT. If the system tablespace is an LMT, you will not be able to create any DMTs in the database. Clearly, Oracle Corp. expects everyone to migrate to LMTs in the near future — quite possibly, DMTs will cease to exist in Oracle 10 — so it would be a smart move to get the migration over and done with before the next version of Oracle arrives. By the way, even if system is an LMT, you will still be able to use the transportable tablespace mechanism to attach a DMT to the database, but that tablespace will have to remain read-only.

As far as LMTs are concerned, performance is pretty much a non-issue. Although the 'amazing' performance benefit of the bitmap management seems to be a commonly touted reason for switching from DMTs to LMTs, it only takes a couple of minutes thought about when, where, and how often you get this benefit to make you realise that it is pretty irrelevant. Just ask yourself — how often should you be allocating and de-allocating space? The correct answer is — hardly ever. Consider the commonest occasions:

- You have allocated a permanent tablespace instead of any form of temporary tablespace as the users' *temporary_tablespace* (an option that is blocked in V9 with error ORA-12911: permanent tablespace cannot be temporary tablespace). Consequently, all sorting, hashing, temporary LOBs and temporary tables get dumped into permanent data segments instead of using the sort extent pool. This could result in a performance problem that could be reduced somewhat by using LMTs, but this isn't an LMT/DMT issue, it is a temporary/permanent issue.

- You have allocated a tablespace of contents type temporary for the users' *temporary_tablespace*, but the extent size you have allocated is extremely small and some sort operations push the extent demand up to tens of thousands, or even hundreds of thousands, of extents. The next time you restart the database, smon runs at 100 percent CPU for ages with a serious blocking effect on most database activity. This could result in a performance hit that could be reduced dramatically by using LMTs, but this is generally an administrative error, not an inherent performance issue. Admittedly, a user's temporary tablespace has to cope with temporary Lobs, temporary tables, sorting and hashing, and these uses may not be compatible. Consequently you may make a deliberate decision to accept this issue — in which case, you definitely do need LMTs for your temporary tablespaces.

- You have created several important, high-volume data objects with a small initial and next extent, and a pctincrease of one (following a well-known and frequently quoted piece of misdirection). Consequently, you have many objects that keep allocating extents because each extent request is for a small extent that is soon filled. Moreover, each request is for an odd-sized extent, and therefore typically requires close to maximum work before Oracle decides how to allocate it. This could result in a performance hit that could be reduced significantly by using LMTs, but it is an administrative error, not an inherent performance issue.
- Your application frequently creates and drops tables on the fly to store transient results. This results in high-stress activity on the space management system. This strategy probably *will* result in a performance hit, but it is a design error, not an inherent performance error. However, for third-party applications in which you can't get the error corrected, this is the one case when the normally marginal performance benefit of LMTs could be a necessary and important damage-limitation exercise (in the short term).

All the above "performance threats" can be avoided, or minimized, without resorting to LMTs, and many DBAs have been taking the necessary steps to avoid them for many years. There is a well-known paper on the topic available through Metalink or OTN ("How to Stop Defragmenting and Start Living"), but in short:

- Don't use storage clauses with objects; always use tablespace defaults.
- Set *pctincrease* = 0 as the tablespace default.
- Set initial = next as the tablespace default.

- Set the minimum extent clause on tablespaces, to match the initial/next.
- Put objects in tablespaces that are appropriate for the expected object size.
- Don't export with compress = y if you plan to recreate tables from an import.
- Make sure you understand the requirements for temporary space, and declare and allocate (multiple) temporary tablespaces accordingly
- Avoid using permanent tables for transient data - look at global temporary tables

So finally, we come to the administrative benefits. Why do LMTs help DBAs keep control of their databases? Essentially, the answer comes back to the paper on defragmenting — the advice it gives is good, and if you switch to uniform-size LMTs, the advice it gives is effectively imposed and enforced at the database level.

If you examine the list above, steps 2, 3, and 4 are covered automatically and unbreakably when you create your tablespaces as locally managed with uniform size — every extent in a tablespace will be the same size. Also, steps 1 and 6 become pretty irrelevant: whatever accidents you have on creating or importing objects with unsuitable storage clauses, Oracle complies with the spirit of the request, but ignores the details of the request by allocating extents according to the tablespace definition. Consequently, much of the strategy that good DBAs have struggled to enforce for many years happens by default with LMTs. And the key word is "struggled." Despite the best efforts of DBAs, it was still possible for things to go wrong with DMTs — with uniform sized LMTs, every extent is forced to be the same size, and no one can break the pattern.

But why is it so convenient to force every extent in the tablespace to be the same size? (And at this point, you may appreciate my earlier comment about avoiding autoallocate LMTs, which allow for half a dozen sizes of extents.) The reasons are, first, ease of monitoring space; secondly, convenience of data packing, and third, reliability of object rebuilds.

Do you have a complicated little script for working out whether or not the next extent for any object in the database will be able to find a large enough space in the right tablespace? If you use uniform LMTs, then this script simplifies to, "Is there any freespace in the tablespace; if so, then it is usable." You could even go so far as to base your reports on a couple of very simple queries:

```
Rem
Rem Find out how many objects per tablespace
Rem Find the unit size for each LMT
Rem Find out the free space per tablespace
Rem
select tablespace_name, initial_extent
from user_tablespaces
where extent_management = 'LOCAL'
and allocation_type = 'UNIFORM'
-- you might include 'SYSTEM'
;
select tablespace_name, count(*)
from dba_segments
group by tablespace_name
;
select tablespace_name, sum(bytes)
from dba_free_space
group by tablespace_name
;
```

By making [some localized variant of] these three queries into /*+ *no_merge* */ in-line views and joining them with a suitable outer join, you could, for example, produce a report showing how many data segments you have per tablespace, and how many of them could extend simultaneously without causing a problem.

Similarly, you could start with a simple query such as:

```
select
     tablespace_name, segment_name, partition_name, extents
from dba_segments
```

You could then combine this with the first of the three queries given previously to capture a daily, or weekly, list showing number of extents per object. This gives you the option for producing a "diff" report of segment growth that can be used to predict future space requirements. Once you have a mechanism that allows you to equate number of extents with size of object, it is so much easier to recognize patterns.

The goal, then, is to ensure that you pick uniform sizes that make it possible to produce warning reports and predictive reports that are useful. And a key feature of usefulness means they should appear only when they have something important to say, and then don't hide it away under a huge volume of trivia and irrelevancy.

The guidelines in the article on defragmentation are good ones to apply to LMTs. Identify objects by such attributes as application, style of use, function, and so on, and finally by size. Typically, you might choose three or four representative sizes for your objects — such as "small," "medium," "large," and "enormous" — and then define tablespaces to match each of these sizes. (I tend to work in multiples of 8 or 16, so for a set of four sizes, and might set up tablespace with uniform sizes of 64K, 512K, 4MB, 16MB). You then allocate objects to tablespace on the basis that fairly static objects should have perhaps 1 to 16 extents, whereas regularly growing objects should add one extent every couple of months. The net effect of this strategy is that you tend to size your database suitably, and don't get any nasty surprises as the data grows.

Of course, mistakes do happen, and your first estimates may put an object in the wrong size of tablespace. It's quite easy to move objects from one tablespace to another — and when you do so, you won't have any problems fitting an object into a tablespace. If there is enough space for an object in a tablespace, then all that space will be usable. Remember the bad old days when you could have 100MB of free space in a tablespace, but be unable to import a 51MB table because the 100MB was made up to two disjoin holes of 50MB each? This simply doesn't happen with LMTs. All holes are the same size, and every object is automatically created as a series of chunks that exactly match the holes. When you move a 24MB table from the "64MB tablespace" to the "1MB tablespace," it doesn't need (and can't have) a single 64MB extent; it automatically arrives as 24 extents of 1MB.

For data warehouse users, the convenience and reliability of space usage also makes it easier to develop a strategy of moving and packing data just before making it read only. (And Oracle 9.2 offers a tremendous added advantage for read-only table data with the compress option for data.) Just before you make a data set (for example, the partitions for last month) read only, you can move and compress the data, rebuild the indexes and trim the containing files to the minimum possible size. And you can do this with a level of convenience and confidence that was not possible with DMTs.

In fact, you could choose to move the objects into tablespaces of the 'wrong' uniform size since there may be an optimum extent count that is a better match for your use of multiple devices, multiple files and parallel execution. The options for proactive space management and performance enhancement through space management become quite interesting when you switch to LMTs.

Conclusion

So what do we really get from LMTs?

- The solution to one special issue that Oracle forces on us because all the function of temporary storage is met by a single tablespace per user.
- Help with avoiding a couple of administrative errors.
- Some important assistance in space monitoring and management.

Should we use LMTs? Absolutely — anything that helps you to eliminate complexity and risk of error, especially in mundane, but time-consuming tasks, is a good thing. And especially if using them introduces a couple of performance-related benefits.

Multiple Block Sizes in Oracle9i

CHAPTER 14

Using Multiple Block Sizes in Oracle9i

The introduction of Oracle9i brought an amazing amount of complexity to the Oracle database engine. Oracle introduced many new internal features, including bitmap free lists, redo log based replication, dynamic SGA, and perhaps the most important feature of all, the ability to support multiple block sizes.

When you strip away all of the advanced features, Oracle's job is to deliver data, and the management of disk I/O is a very critical component and tuning of any Oracle database. Anything that can be done to reduce the amount of disk I/O is going to have a positive impact on the throughput of the Oracle database system.

If we take a look at the various tuning activities within Oracle database, will see that the common goal of almost Oracle tuning has the directed and immediate goal of reducing disk I/O. For example, tuning an SQL statement to remove a full table scans makes the query run faster because of the direct reduction in the amount of data blocks that are read from the disk. Adjusting instance tuning parameters such as *db_cache_size* also has the goal of reducing the amount of disk overhead.

To understand how using multiple block sizes can improve performance of the Oracle database we first have to start by taking a look at the basic nature of disk I/O. Anytime an

Oracle data block is accessed from disk, we commonly see three sources of delay. The first and most important source of delay is the read-write head movement time. This is the time required for the read-write head to position itself under the appropriate cylinder. We also see rotational delay as the read-write head waits for the desired block the past beneath it, and the third source of delay is the data transmission time from the disk back to the Oracle SGA.

If we accept the premise that 99 percent of the latency is incurred prior to actually accessing the desired data block, then it makes sense that the marginal cost for reading a 32K block is not significantly greater than the cost of reading a 2K block. In other words, the amount of disk delay is approximately the same regardless of the size of the block. Therefore it should follow that the larger the block that you can read in on a single I/O, the less overall I/O will be performed on the Oracle database.

The principal behind caching is not unique to Oracle databases. Access for RAM is measured in nanoseconds, while access from disk is generally measured in milliseconds. This amounts to a to an order of magnitude improvement in performance if we can get the Oracle data block into a RAM buffer. As Oracle grows more sophisticated and RAM becomes cheaper, we tend to see Oracle9i databases with system global areas (SGA) that commonly exceed 10 GB. This has important ramifications for the performance of the Oracle database because once read, the Oracle data blocks reside in RAM where they can be accessed tens of thousands of times faster than having to go to disk in order to retrieve the data block.

RAM buffers and Oracle data access Oracle has always provided RAM data buffers to hold incoming data blocks, and data can be read from the buffers 14,000 times faster than reading the data block from disk. The RAM data buffer has evolved from a single buffer in Oracle7 to three data buffers in Oracle8i. These were known as the KEEP pool, the RECYCLE pool, and the DEFAULT pool (refer to figure 1).

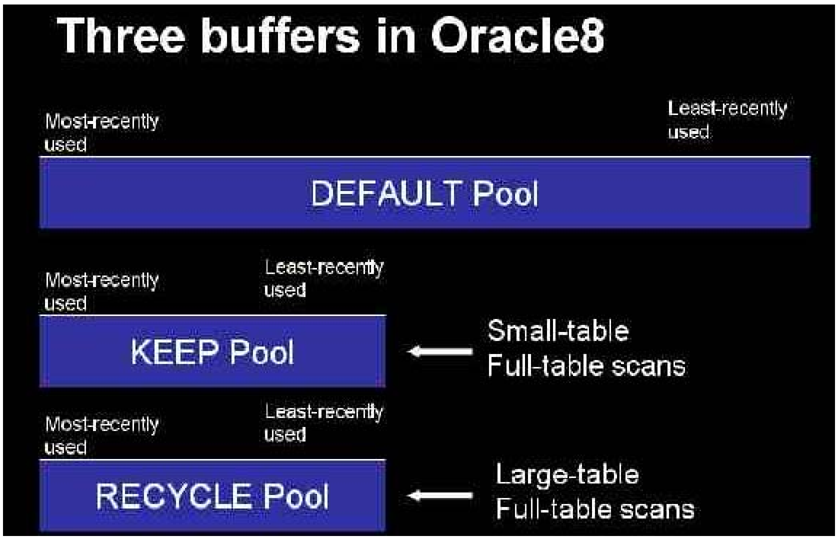


Figure 1 - *The Oracle8 data buffers*

In Oracle9i we still have the three data buffers, but we also have the ability to create a data buffer for every supported blocksize for the Oracle server (refer to figure 2).

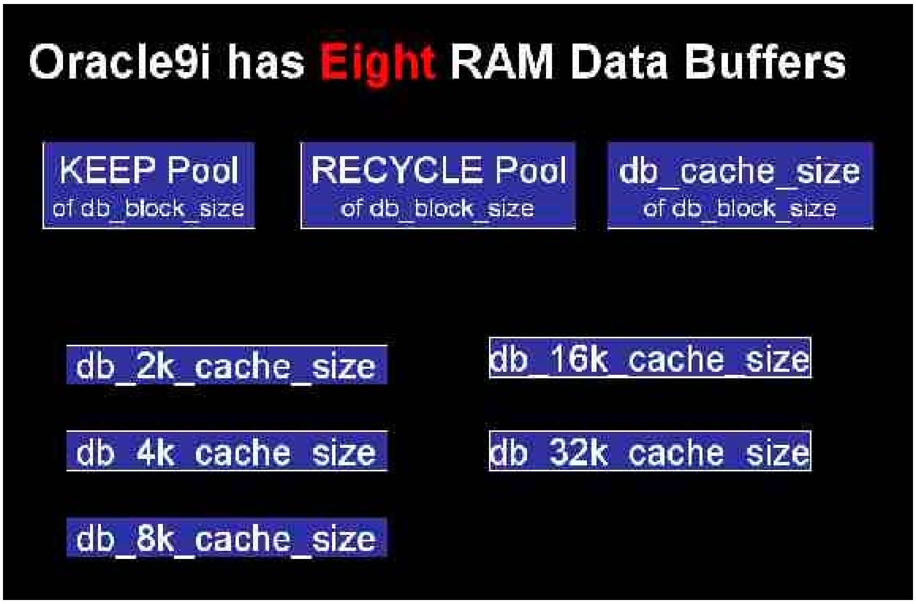


Figure 2 - *The eight data buffers for Oracle9i*

Within each data buffer, the data buffer hit ratio measures the propensity of a data block to be in RAM memory. It is the job of the Oracle administrator to allocate RAM pages among the data buffers to ensure the optimal amount of RAM caching. With small buffers, a marginal increase of pages results in superior caching (refer to figure 3).

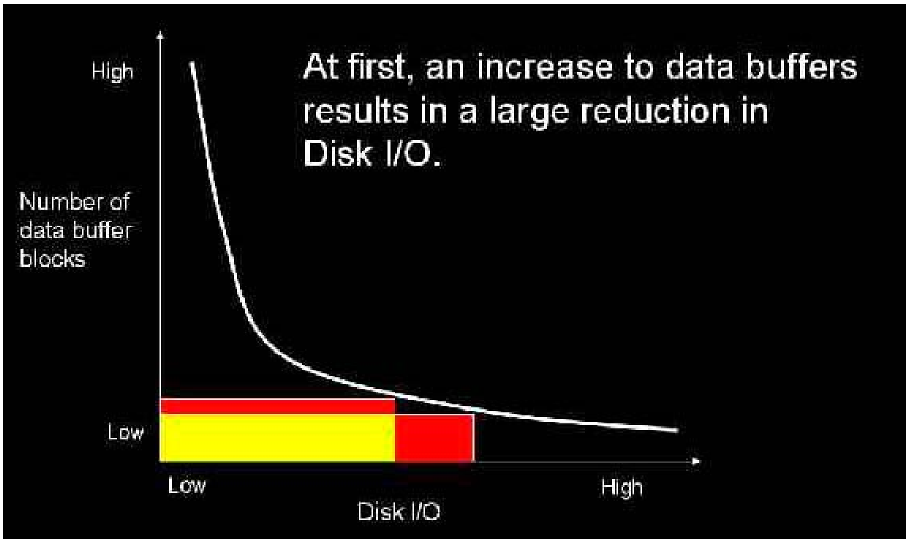


Figure 3 - *RAM pages added to a small data buffer*

As the RAM cache is increased, the marginal benefit from adding pages decreases (refer to figure 4).

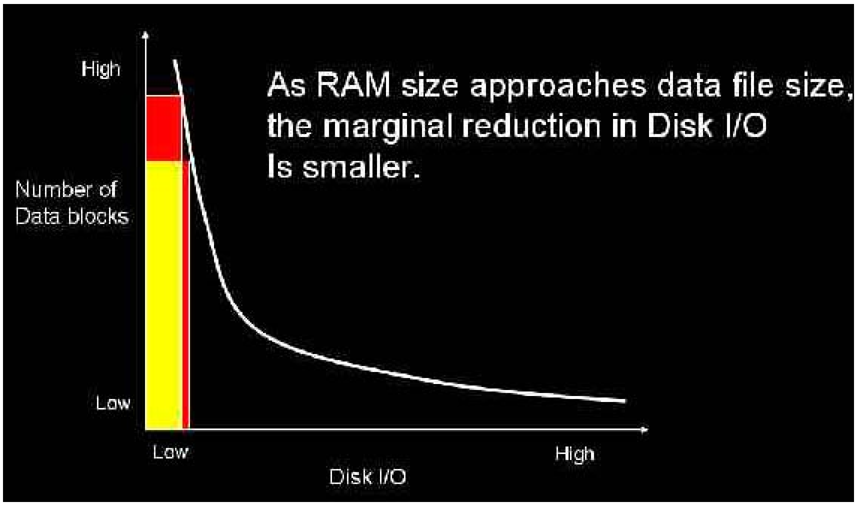


Figure 4 - *The marginal decrease of disk I/O with large data buffers*

Indexes and Large Data Blocks

Prior to Oracle9i, Oracle professionals noticed that by moving the entire database to a larger block size, they reduce disk I/O improve the performance of the entire system. This is somewhat counterintuitive, and people ask "if I only need an 80-byte row, where do I get the benefit of reading 16K block?"

The answer has to do with indexes. Most well-tuned Oracle database have index based roughly equal to the space of the table data. There's no question of a large block size for indexes is going to reduce I/O, and therefore improve the overall performance of the entire database.

Hence, one of the first things the Oracle9i database administrator will do is to create a 32K tablespace, a corresponding 32K data buffer, and then migrate all of the indexes in their system from their existing blocks into the 32K tablespace. Upon having done this, the Oracle9i database can read a significant amount of index note branches in a single disk I/O, thereby reducing stress on the system and improving overall performance.

Allocating Objects into Multiple Block Buffers

So given that we have the ability to create multiple data buffers within the Oracle database, how do we decide what data that we want to put each of these data buffers?

Let's start with some of the more common techniques.

Segregate large-table full-table scans - Tables that experience large-table full-table scans will benefit from the largest supported block size and should be placed in a tablespace with your largest block size.

Set *db_recycle_cache_size* carefully - If you are not setting *db_cache_size* to the largest supported block size for your server, you should not use the *db_recycle_cache_size* parameter. Instead, you will want to create a db_32k_cache_size (or whatever your max is), and assign all tables that experience frequent large-table full-table scans to the largest buffer cache in your database.

The Data Dictionary uses the default cache - You should ensure that the data dictionary (e.g. your SYSTEM tablespace) is always fully cached in a data buffer pool. Remember, the block size of the data dictionary is not as important as ensuring that the data buffer associated with the SYSTEM tablespace has enough RAM to fully-cache all data dictionary blocks.

Segregate Indexes - in many cases, Oracle SQL statements will retrieve index information via an index range scan, scanning the b-tree or bitmap index for ranges of values that match the SQL search criteria. Hence, it is beneficial to have as much of an index residing in RAM as possible. One of the very first things the Oracle 9i database administrator should do is to migrate all of their Oracle indexes into a large blocksize tablespace. Indexes will always favor the largest supported blocksize.

Segregate random access reads - For those databases that fetch small rows randomly from the disk, the Oracle DBA can segregate these types of tables into 2K Tablespaces. We have to remember that while disk is becoming cheaper every day, we still don't want to waste any available RAM by reading in more information to RAM number actually going be used by the query. Hence, many Oracle DBAs will use small block size is in cases of tiny, random access record retrieval.

Segregate LOB column tables - For those Oracle tables that contain raw, long raw, or in-line LOBs, moving the table rows to large block size will have an extremely beneficial effect on disk I/O. Experienced DBAs will check *dba_tables.avg_row_len* to make sure that the blocksize is larger than the average size. Row chaining will be reduced while at the same time the entire LOB can be read within a single disk I/O, thereby avoiding the additional overhead of having Oracle to go out of read multiple blocks.

Segregate large-table full-table scan rows - When the recycle pool was first introduced in Oracle8i, the idea was the full table scan data blocks, (which are not likely to be re-read by other transactions), could be quickly flushed through the Oracle SGA thereby reserving critical RAM for those data blocks which are likely to be re-read by another transaction. In Oracle9i, you can configure your recycle pool to use the a smaller block size.

Check the average row length - The block size for a tables' tablespace should always be greater than the average row length for the table (*dba_tables.avg_row_len*). Not it is smaller than the average row length, rows chaining occurs and excessive disk I/O is incurred.

Use large blocks for data sorting - Your TEMP tablespace will benefit from the largest supported blocksize. This allows disk sorting to happen in large blocks with a minimum of disk I/O.

Tools for Viewing Data Buffer Usage

The process of segregating Oracle objects into separate data buffers is fairly straightforward and Oracle9i provides tools to assist in this effort. Many Oracle administrators are not aware of those table blocks which consume a disproportional amount of data space within the data buffer caches, and Oracle9i provides numerous scripts to allow you to see which objects reside most frequently within the data cache.

The query below counts the number of blocks for all segments that reside in the buffer cache at that point in time. Depending on your buffer cache size, this could require a lot of sort space.

```
column object_name format a40
column number_of_blocks format 999,999,999,999
column object_name      format a40
column number_of_blocks format 999,999,999,999

SELECT
   o.object_name,
   COUNT(1) number_of_blocks
FROM
   DBA_OBJECTS o,
   V$BH bh
WHERE
   o.object_id  = bh.objd
AND
   o.owner != 'SYS'
GROUP BY
   o.object_name
ORDER BY
   count(1) desc;
```

Here we see the object name and the number of data blocks in the buffer.

```
OBJECT_NAME                              NUMBER_OF_BLOCKS
---------------------------------------- ----------------
ORDER_TABLE                                       123,273
ORDER_IDX                                         112,492
CUSTOMER                                           83,272
. . .
OEM_EXT                                               701
```

Creating Separate Data Buffers

The process of assigning table or index blocks into named block size tablespaces is very straightforward within Oracle9i. We begin by creating a tablespace and using the new *blocksize* parameter in the create tablespace syntax. In the example below we create a 32K tablespace within the Oracle database.

```
create tablespace
   32k_tablespace
datafile
   '/u01/oradata/mysid/32k_file.dbf'
size
   100M
blocksize
   32k
;
```

Once we have the names tablespaces created, the next step is to set a database cache to correspond to that block size. Remember, and Oracle 9i we no longer have the *init.ora* file, and we create the named cache dynamically with an alter database statement.

```
alter system set db_2k_cache_size=200M;
alter system set db_4k_cache_size=500M;
alter system set db_8k_cache_size=800M;
alter system set db_16k_cache_size=1600M;
```

Once we've created the named RAM buffer, and the tablespace, we are now ready to migrate the Oracle objects into the new tablespace. There a variety of methods for moving objects from one tablespace to another, and many Oracle administrators are already familiar with using the create table as select or CTAS syntax in order to move the table. For indexes, the alter index rebuild command can be used to quickly migrate an index tree between tablespaces.

Conclusion

Out of all of the sophisticated enhancements of Oracle9i, many experienced DBAs consider multiple block sizes to be the most important feature for tuning the Oracle database. The Oracle database administrator now has up to seven separate and distinct data pools that they can manage a control, giving the Oracle9i DBA a huge amount of control over the amount of data buffer blocks that can be assigned to specific database objects. Through judicious inspection all of buffer access characteristics, overall disk I/O can be tremendously reduced, and the performance of the database will be greatly improved.

Automated Table Reorganization in Oracle8i

CHAPTER 15

Automated Table/Index Reorganization In Oracle8i

Automation can free the DBA of boring, time-consuming tasks and allows him to focus on more challenging activities.

Databases normally have a number of very volatile tables and indexes, and I felt that strong, automated reorganizations would be beneficial. The result is a comprehensive solution - a complete PL/SQL package that can perform periodic table and associated indexes reorganization automatically, is self-tuning, portable, and (almost) platform and version independent. I started this project since similar commercial products are a lot more complex and normally are extremely expensive.

This PL/SQL package is a complementing solution to the one presented in my article, "Setting Up an Automated Index-Rebuilding System" (Oracle Publishing Online - September 2001). It can be run as a periodic complement to the auto-re-indexing package (PKG_NDXSYS), or instead of it. This solution has been tested on Unix (HP 10.7 and 11, Sun Solaris 7 and 8, AIX 4.3, Linux 2.x) and Windows servers (NT4, 2000), on Oracle versions 8.1.5, 8.1.6, 8.1.7. Should work just fine with Oracle 9i, but not with versions earlier than 8.1.5. It requires some knowledge of UNIX shell scripts, SQLPlus scripts, and PL/SQL. However, the full scripts are provided and minimal knowledge would be enough to install the package and get started.

When Reorganizing, How Many Extents to Use?

The current view is that objects do not need to be compressed into a single larger extent in order to have good performance. Although once the recommendation was to have single-extent objects, today Oracle recommends not to have more than 1024 extents per object and that an object with reasonably and equally sized extents is leading to the best performance. A very interesting method is described in the Oracle white paper 711, "How to Stop Defragmenting and Start Living." Since there is no formula that I know of, and after some experimentation, I propose the use of an algorithm for table and index rebuilds, done manually or automatically (algorithm described in the comments within the package). The extent size upper limit can be increased for extremely large objects.

Refer to the Oracle manuals for the official view on space management: "*Oracle8i Tuning Manual*" (Tuning I/O - Avoiding Dynamic Space Management - Evaluating Multiple Extents), and "*Oracle8i Backup And Recovery Manual" (Developing A Backup And Recovery Strategy - Developing A Backup Strategy - Perform Backups After Unrecoverable/Unlogged Operations*").

NOTE: The Unrecoverable/Nologging option should not be used if there is a standby database.

Possible Reorganizing Strategies

- Cron jobs at fixed times - the most common.
- Inside batches, after massive changes, to keep objects current - fairly common.

- Inside batches, before massive changes, to improve the batch performance - less used.
- Just before backups, to backup an optimized database, or just after backups, to backup faster the database.
- Dedicated systems, with collection tables, control procedures, etc. - in more complex environments.

Assumptions and Experimental Figures

- On average, roughly, we could rebuild in 1 hour 5,000,000 rows or 5 GB
- We had a time window of between 1 and 3 hours, between 21:00 and 24:00
- We could not do all objects in one day (session)
- Time acceptable without reorg is within a certain limit
- Some days cannot be used for reorg, being used for cold backup, etc.

Some Procedures Related to Table Reorganization

- Coalesce free extents in tablespaces, before and after each reorg - here is a script - *ts_coalesce.sql* (http://www.dbazine.com/code/ts_coalesce.sql.txt)
- Build a fragmented tablespace as a test environment - use a script like *ts_fragment.sql* (http://www.dbazine.com/code/ts_fragment.sql.txt) (first create a user "cbosys2" and a tablespace for it)
- The only visible objects in a GUI tablespace tool like Tablespace Manager (Map) are the ones that actually take up physical space: tables, indexes (regular, primary keys, unique

constraints, etc.) If some objects are not visible, it means they are just references or definitions: (foreign keys, not nulls, checks, etc.)

- Determine the fragmentation level in a database - here are some fragmentation assessment criteria:
    - high numbers of extents (acceptable < 1024 extents for very large objects - look out for extents per object > 5)
    - high percentages of chained rows per table (acceptable < 3 percent- look out for percentages > 0.1 percent) - analyze the tables first
    - high percentages of free space inside blocks (look out for FREESPACE/BLOCK > 2*PCTFREE)
    - high percentages of free space above highwatermark (look out for EMPTY BLOCKS ABOVE HWM > 50 percent)
- here are a few scripts to help with these tests: *objects_for_reorg.sql* (http://www.dbazine.com/code/objects_for_reorg.sql.txt) and *obj_next_ext_fail.sql* (http://www.dbazine.com/code/obj_next_ext_fail.sql.txt)

Important Issues Regarding Table/Index Moving/Rebuilding

- You cannot perform table reorg without index reorg, even if you do not re-structure or relocate the index, because the index becomes UNUSABLE after the table reorg, as the ROWID references become invalid. That is why if you have a table reorg scheduled, you may skip a scheduled index reorg (for the affected indexes).

- The new index is built either from the data in the index, or from the data in the table, whichever source is smaller. This is called "fast rebuild" and is available since Oracle 7.3.4. If you suspect the index is already corrupted, you will have to drop the index and re-create it with fresh data from the table.
- Oracle will place a lock on the table for the duration of the table/index move/rebuild. The lock affects INSERT, UPDATE, DELETE statements, but allows SELECT statements. The DML will have to wait until the move/rebuild is done. However, Oracle 8i (8.1.x) can allow any DML statement if the DDL runs the ONLINE option.

  ```
  ALTER TABLE/INDEX table_name/index_name MOVE/REBUILD ONLINE;
  ```
- In this situation (locking), the indexes may not be available to users for some periods of time and performance may be affected. Conversely, the move/rebuild will fail if somebody else has put a lock on the table and Oracle cannot acquire exclusive access.
- While ANALYZE COMPUTE does not lock the object, the ANALYZE VALIDATE STRUCTURE locks the object the same as the ALTER TABLE/INDEX MOVE/REBUILD.

The Behavior of the "Alter Table/Index Move/Rebuild" Commands

```
ALTER TABLE/INDEX table_name/index_name MOVE/REBUILD TABLESPACE
tablespace_name
STORAGE (PCTINCREASE 0 INITIAL 512M NEXT 256M);
```

will cause the database to try and locate an extent of 512M in the selected tablespace, to allow rebuild and compression of the existing object. If the object is larger than 512M, the rebuild process will try to acquire a next extent of 256M and continue the rebuild. If there is no extent of 512M, on most versions and platforms, the rebuild process will revert to the tablespace default for INITIAL, and start the rebuild (check or experiment with your version to determine how this feature works). Normally, this does not fail. However, the free space for the next extent (256M) has to be available and found or the rebuild will fail.

If you are unsure, the clause STORAGE (INITIAL 0K NEXT 0K) will often revert to tablespace defaults and almost always work successfully, if the total free space is enough, but you can end up having a large number of extents (even hundreds or thousands).

Limitations of the "`ALTER TABLE MOVE`" Command:

- Supported only in Oracle 8.1.5 and higher
- Does not support directly some objects and some data types:
    - Clustered tables, IOT's, overflow table of an IOT, hash and composite partitions (range partitions are

The Behavior of the "Alter Table/Index Move/Rebuild" 155

supported), tables with columns containing LONG and LONGRAW types, tables with columns containing user-defined types, indexes on such columns, function-based indexes, domain indexes, partitioned tables containing a LOB column can be reorg'ed on a per partition only, partitioned indexes may not be rebuilt as a whole, for some object types and data types there are special commands that can be used as workarounds

- Most of these limitations apply also to CTAS (Create Table As Select) methods
- Some of them apply also to index rebuilds
- You can still use the `SQL*Plus COPY` command or the EXPORT/IMPORT utilities

Manual Object Reorganization

Roughly, for us, the execution time was 100 minutes per 1 GB of really used space (data). Resources: reorganizing can take up to 300MB of memory and up to 30 percent CPU. It takes a lot less on smaller systems.

- *reorg.sql* - script to reorg all tables in the database (http://www.dbazine.com/code/reorg.sql.txt)
- *reindex.sql* - script to rebuild invalidated indexes - called by *reorg.sql* (http://www.dbazine.com/code/reindex.sql.txt)
- *ts_coalesce.sql* - script to coalesce tablespaces - called by *reorg.sql*

This method will keep the rest of the database online and available to users. For each table there are two steps.

Step 1

The ALTER TABLE MOVE command will lock the table for changes, but will allow queries. While the table is moved, the new table will actually be a TEMPORARY segment in the destination tablespace, named something like "52.42" for the duration of the reorg. The old table will continue to be there and is dropped (and the new table renamed to the old name) only when the new table build is finished successfully. The TEMP tablespace is normally not used. However, RBS and redo logs can take a serious hit.

If there is not enough space, the procedure will fail and the old table will remain in place. This procedure can be run by the schema owner or by the SYSTEM user. Relocating tables to other tablespaces can be done manually, by editing the generated *reorg.lst* script. If there is enough spare space, one can create one or two flip-flop tablespaces, dedicated to moving around reorganized objects, so that the objects are always rebuilt in only a few larger extents when moved to the other tablespace.

Step 2

The table move will change the ROWID's of the table rows, and as such the indexes, which are based on ROWID's, will become invalid (UNUSABLE). Therefore, the need to execute step two immediately after step one - rebuild the invalid indexes on the current table.

At the same time an advantage of using the table move procedure is all constraints are preserved, and index definitions are also saved, so that reindexing is possible using the fast index REBUILD method, rather than the slower index DROP and CREATE method.

The ALTER INDEX REBUILD command will restore the index to a valid state. While the index is rebuilt, the new index will actually be a TEMPORARY segment in the destination tablespace, named something like "15.64" for the duration of the rebuild. The old index will continue to be there and is dropped (and the new index renamed to the old name) only when the new index is finished successfully.

There is also another type of TEMPORARY segments during the rebuild: the segments for storing the partial sort data, because for larger indexes the *sort_area_size* is normally too small. These segments are located in the TEMP tablespace and they become visible as soon as the *sort_area* is filled and spills over to disk. When the whole index is contained in these segments, their growth will stop and the segments that will hold the final index will start to grow in the destination index tablespace. For small indexes, there are no segments in the TEMP tablespace, as the sorting happens in memory (in the *sort_area*, outside the SGA). Anyway, especially for large objects, RBS and redo logs can take a serious hit. You should also watch for space in the ARCHIVE LOGS directory.

If there is not enough space, the procedure will fail and the old index will remain in place. This procedure can be run by the schema owner or by the SYSTEM user. Relocating indexes to other tablespaces can be done manually, by editing the generated *reindex.lst* script. If there is enough spare space, one can create one or two flip-flop tablespaces, dedicated to moving around reorganized indexes, so that the indexes are always rebuilt in only a few larger extents when moved to the other tablespace.

This method is by far the preferred manual method for table/index relocation and reorganization/defragmentation.

However, I would not recommend running these scripts against the whole database in one session.

If you need more sessions to go through all the objects in the database, you can use a similar technique to the one illustrated for building session-based scripts for the ANALYZE command, in my article *Automated Cost Based Optimizer* (Oracle Publishing Online - September 2000) - section "Manual Analysis of the DB1 Database".

Automated Object Reorganization

Our strategy will be a combination of cron jobs and a PL/SQL package (PKG_TABSYS). Reorganizing tables/indexes normally can be done online, without dropping objects, and has a very positive impact on the general performance of the database. I have been running the package for the last year with no serious problems. The execution times seem to decrease steadily after a few runs, as the package has some self-tuning capability. The average move/rebuild times on Oracle8i have come down from 90 minutes to 45 minutes. In theory, at least, the more it runs, the less fragmented the objects become, and the faster the systems will be. Some degree of tablespace level fragmentation is to be expected. Remember that tablespace fragmentation does not affect performance, but only the growth capacity of the objects (specially very large ones). You should keep an eye on the free space.

Prerequisites

- You should have some system privileges (see the beginning of the *install_tabsys.sql* script http://www.dbazine.com/code/install_tabsys.sql.txt).
- Set *utl_file_dir* = * (or at least c:\temp, or /tmp, etc.) in *init.ora*, in order to allow log files to be created
- Set *job_queue_processes* = 2 (or higher) in *init.ora*, in order to allow *dbms_job* scheduling to work

Associated Tables

A set of three tables (*tabsys_list*, *tabsys_sort*, and *tabsys_hist*) hold identifying, processing and historical information. The data collected in the history table can also be used for queries later on to find information useful for growth monitoring and capacity planning. A fourth table (*tabsys_ts*) holds the information about corresponding pairs: source table tablespaces and target table tablespaces. You may want to give careful consideration to this table. This section will cause table relocation. Check for available space in the tablespaces.

Search for the following section in the *install_tabsys.sql* script and adapt to your particular environment, BEFORE installing the package. The package reads this table and checks object location every time it runs.

```
-------------------------------------------------------------------
--------
prompt POPULATING TABLE tabsys_ts WITH YOUR VALUES
prompt
TRUNCATE TABLE tabsys_ts;
COMMIT;
INSERT INTO tabsys_ts VALUES ('SYSTEM', 'USERS');
COMMIT;
```

This will relocate any table found in any of the tablespaces in the left-hand column (SYSTEM, etc.) to the corresponding tablespace in the right-hand column (USERS, for objects not owned by SYS or SYSTEM, in this case). If you do not populate the table or just insert the same values left and right, then the object will not be relocated. You can update this table manually any time in the future.

Overview of the Package

Basically, the Automated Table/Index Rebuild package (PKG_TABSYS) runs the 'ALTER TABLE MOVE' command followed immediately by the 'ALTER INDEX REBUILD' command, and will also:

- Clean up residual temporary segments
- Coalesce free space in tablespaces
- Analyze the structural integrity of the objects
- Generate valuable statistics usable by the CBO
- Deallocate unused space from object blocks
- Shrink object segments
- Re-align the highwatermark to low levels
- Re-organize fragmented objects into fewer extents
- Re-structure (optimize) tablespace storage options
- Re-structure (optimize) table storage options
- Compact table blocks into fewer blocks
- Re-attempt to run with modified parameters in case of failure
- Generate alerts if it detects failure to grow or reorg

- Detect some generic unavailability conditions
- Process both tables and indexes
- Reorganize/defragment, actually, the entire database

The code (circa 2500 lines) performs a lot of error checking and decision making in support of the commands. Since you cannot reorg everything in one session, objects are sorted and organized in manageable sessions, which are then run one a day, until the cycle is finished and a new cycle begins. Each table reorg will cause the associated indexes to become invalid (UNUSABLE) and as such an index rebuild MUST be performed after the table reorg.

Initially, we build a few tables (see section ASSOCIATED TABLES), then we populate them with data from the DATA DICTIONARY and calculated from running the package, with information about the processable objects (tables and indexes), sorted by size (bytes) in descending order. The system examines the objects one by one and marks them with 0 if no reorg needed, with 99 if reorg required, with 999 if last reorg failed, and with 9999 if last reorg was successful.

Based on a series of rules, the system then decides which object is assigned to which session. It starts with the first session, 'empty', and examines the first object against the rules. If there is a need for reorg, the object is assigned to the current session, if there is no match, it is left for the next session. The process continues until all objects are assigned, and there is a number of sessions.

We then start to run the sessions, one at a time (probably daily). The results of the run are written back into our 'TABSYS' tables, to be used the next time we build sessions. When all sessions are done, we examine the logs in the /tmp or c:\temp directories for failed runs, and attempt to run them again. Upon completion, an email message is sent to the DBA and the process is ready to start again.

When run manually in an SQLPlus session, display procedures ensure that debugging and detailed logging (hundreds of lines of messages) are made as easy as possible - currently these modules are commented out to avoid crashing the package because of overloading the server output buffer - uncomment them selectively for databases with very large numbers of objects.

Although it will not account for all situations, the package does log a wide variety of errors. The DBA will treat errors manually as the automated system will only try to re-run a session in case of failure. Some errors, like "failed because of resource busy", simply mean that a lock could not be obtained, as some other process was using the object, and can be ignored, as it will probably succeed on the next run. A number of conditions and options (e.g. parallel, analyze, nologging, etc.) are also available to be enabled or disabled in the package body. Objects dropped after the list was created will also cause benign errors. Also, hitting tables with data types not supported for MOVE will simply generate an error message and skip to the next object. If the package is run automatically with 'DBMS_JOB', we get only the *summary output* (http://www.dbazine.com/code/DB1-TABsysPKG.log.txt), which can include captured error messages. Most error messages will also be logged in the 'TABSYS' tables themselves.

Setup

The package is installed into the default Oracle schema 'MHSYS', which I use to host my automation packages. IT CAN BE INSTALLED, AS IS, FOR UNIX AND NT BASED SERVERS. It is a pretty comprehensive piece of software, which is compatible with Oracle 8.1.5 or higher, on both UNIX and NT, and includes routines to detect the current OS, Oracle version and SID,

The code is amply commented. Run the *install_tabsys.sql* script as user 'SYSTEM' from SQLPlus. Before installing, read the top of the package body, just in case you need to make some modifications. This section can also be used for tuning later, by changing the values of a very large number of constants. Make sure the script does not drop the existing schema 'MHSYS' if already installed. The defaults will cover most situations and, most likely, nothing will need to be changed. It has been run against objects with sizes of up to 3500 MB. Sessions can vary between 10 - 300 minutes. Have the logs emailed to you or, at least, examine them manually.

You can use scripts to schedule or run the package similar to the ones described in my article, "Setting Up an Automated Index-Rebuilding System" (Oracle Publishing Online - September 2001).

CHAPTER 16

Using External Table in Oracle9i

External Tables in Oracle9i

Here's a step-by-step example of creating an external table and querying the data source from within Oracle along with a discussion of practical applications for external tables, performance and management issues.

ORACLE9i has many new features, and one of my favorites is the ability to create external tables. An external table is a table whose structure is defined inside the database even though its data resides externally as one or more files in the operating system (see Figure 1). External tables are very similar to regular tables in Oracle, except the data isn't stored in Oracle datafiles and isn't managed by the database.

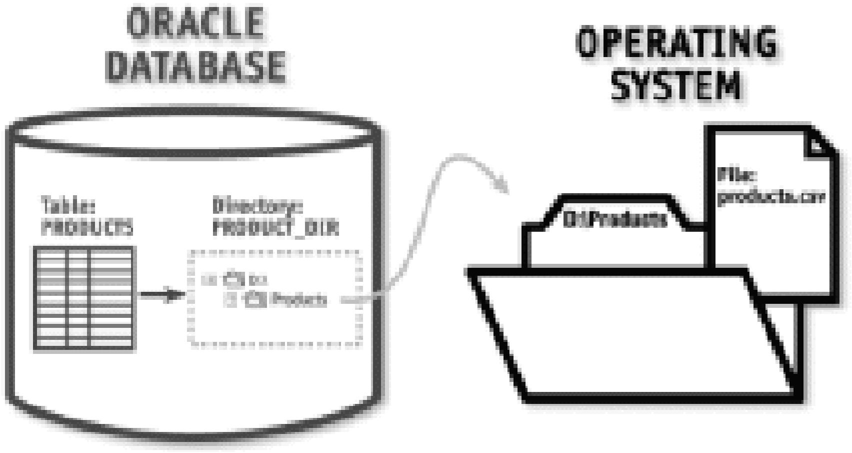
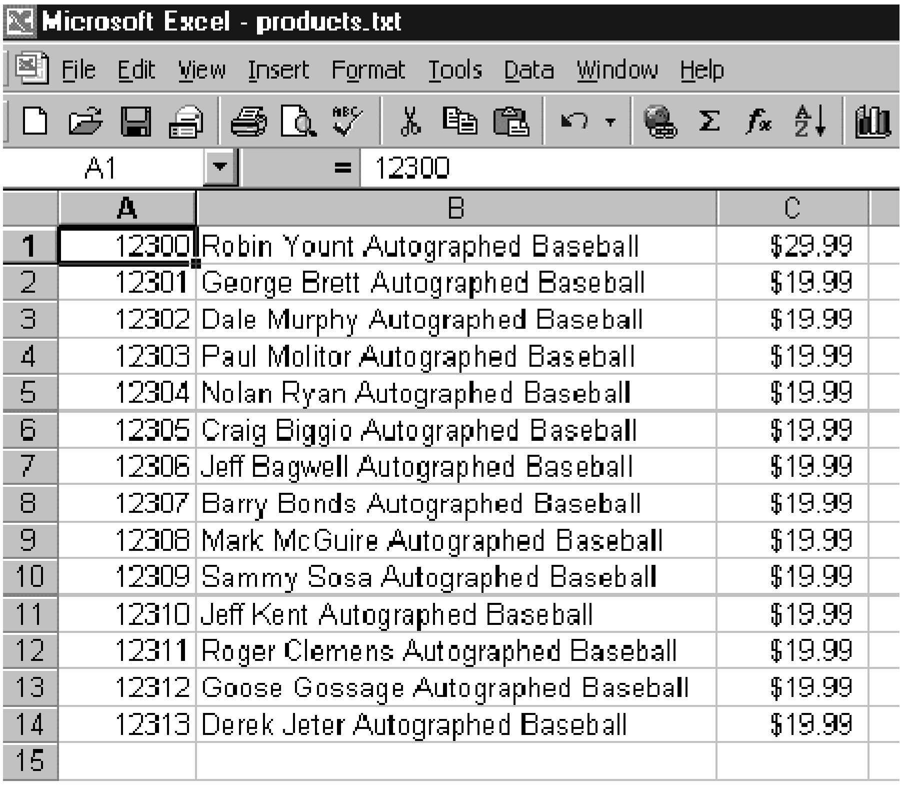


Figure 1: *External table structure in Oracle.*

Example

This example begins with product information listed in a Microsoft Excel spreadsheet (see Figure 2). The data is saved in comma-separated values (CSV) format to D:\products\products.csv. The spreadsheet contains three columns: Product Number, Description, and Price. This file contains the data that we'll query from Oracle.


Microsoft Excel - products.txt
File Edit View Insert Format Tools Data Window Help
A1
= 12300

	A	B	C
1	12300	Robin Yount Autographed Baseball	$29.99
2	12301	George Brett Autographed Baseball	$19.99
3	12302	Dale Murphy Autographed Baseball	$19.99
4	12303	Paul Molitor Autographed Baseball	$19.99
5	12304	Nolan Ryan Autographed Baseball	$19.99
6	12305	Craig Biggio Autographed Baseball	$19.99
7	12306	Jeff Bagwell Autographed Baseball	$19.99
8	12307	Barry Bonds Autographed Baseball	$19.99
9	12308	Mark McGuire Autographed Baseball	$19.99
10	12309	Sammy Sosa Autographed Baseball	$19.99
11	12310	Jeff Kent Autographed Baseball	$19.99
12	12311	Roger Clemens Autographed Baseball	$19.99
13	12312	Goose Gossage Autographed Baseball	$19.99
14	12313	Derek Jeter Autographed Baseball	$19.99
15			

Figure 2: *Product data in Excel.*

After saving the file from Excel, the next task is to create a DIRECTORY object in Oracle that points to the physical operating system directory that contains the file. This DIRECTORY is required in order to create the external table.

```
SQL> CREATE DIRECTORY PRODUCT_DIR AS 'd:\products';
Directory created.
```

Now the external table is created by using the CREATE TABLE command:

```
create table products (
product_no number,
description varchar2(100),
price varchar2(20)
)
organization EXTERNAL (
type oracle_loader
default directory PRODUCT_DIR
access parameters
( records delimited by newline
badfile 'products.bad'
logfile 'products.log'
fields terminated by ','
)
location ('products.csv')
)
reject limit unlimited
/
```

The first part of the CREATE TABLE statement holds no surprises. Notice, however, that the next part of the statement specifies ORGANIZATION EXTERNAL, which indicates that this table is an external table. This part of the statement also specifies a type of oracle_loader-the only one currently supported by Oracle. Oracle_loader is actually an oracle TYPE object defined in the database to handle the processing. Also notice that the directory object is part of the CREATE TABLE statement; it tells Oracle where to find the files.

The next part of the statement specifies the access parameters, which should look familiar to anyone who's experienced with SQL*Loader:

- *records delimited* by specifies the characters that will be used to separate rows.
- *badfile* specifies the file that Oracle will use to store the rejected rows.
- *logfile* specifies the file that Oracle will use to store log information. Documentation of any errors will be provided in this file.
- *fields* terminated by specifies the field separator that will distinguish one column from another during the load.

Finally, the location and reject limit are specified:

- *location* provides the name of the actual file to access. If Oracle needs to access multiple files, they can be specified as follows:

  ```
  location ('file1.dat', 'file2.dat')
  ```

- *reject limit* specifies the number of rows that can be rejected before the command returns an error. If this threshold is reached, the following error appears when trying to access the table:

  ```
  ERROR at line 1:
  ORA-29913: error in executing ODCIEXTTABLEFETCH callout
  ORA-30653: reject limit reached
  ORA-06512: at "SYS.ORACLE_LOADER", line 14
  ORA-06512: at line 1
  ```

The DDL for creating the statement will run even if the file doesn't exist in the system, which can produce mixed results. On the one hand, you won't know whether the table was successfully created until a statement is executed against the table, which in a data-warehousing environment might be at 3:00 a.m. Conversely, the file doesn't have to exist at the time the table is created. In fact, the file can come and go as needed, which is quite customary in OLAP environments.

The external table is now created. However, if another user tries to access the table at this point, that user will receive an error:

```
SQL> select count(*) from dave.products;
select count(*) from dave.products
                          *
ERROR at line 1:
ORA-06564: object PRODUCT_DIR does not exist
```

To prevent this error, you must grant read and write access on the directory for any user who wants to select data from the table. Granting SELECT on the table itself will allow the object to be seen, but you must also grant access to the underlying directory object.

```
grant read, write on directory products_dir to alex;
```

Listing 1: *Querying the table.*

```
SQL> select product_no, substr(description,1,40) "Desc", Price from
products;

PRODUCT_NO Desc PRICE
---------- -------------------------------------- --------------
12300 Robin Yount Autographed Baseball $29.99
12301 George Brett Autographed Baseball $19.99
12302 Dale Murphy Autographed Baseball $19.99
12303 Paul Molitor Autographed Baseball $19.99
12304 Nolan Ryan Autographed Baseball $19.99
12305 Craig Biggio Autographed Baseball $19.99
12306 Jeff Bagwell Autographed Baseball $19.99
12307 Barry Bonds Autographed Baseball $19.99
12308 Mark McGuire Autographed Baseball $19.99
12309 Sammy Sosa Autographed Baseball $19.99
12310 Jeff Kent Autographed Baseball $19.99
12311 Roger Clemens Autographed Baseball $19.99
12312 Goose Gossage Autographed Baseball $19.99
12313 Derek Jeter Autographed Baseball $19.99

14 rows selected.
```

Read/write access means that Oracle will be allowed to write to that directory when it needs to update the logfile or badfile. As an OS user, you don't have access to those files in the operating system unless your ID has proper privileges; as a result, security isn't compromised.

After creating the external table and granting privileges, the table can be queried like any other table (see Listing 1).

The external table can be used as a substitute for SQL*Loader and a regular table can be used to hold its data:

```
INSERT INTO PROD.PRODUCTS AS SELECT * from DAVE.PRODUCTS;
```

The data that was in Excel is loaded into Oracle, which allows it to be backed up and to perform better than an external table.

Limitations

External tables in Oracle9i have the following limitations:

- They're read-only, so no data manipulation language (DML) operations (such as Insert, Update, or Delete) can be performed against them. Also, no indexes can be defined on the table. Oracle does plan to support writing to these tables in a future release.
- They don't support files larger than 2GB. If you attempt to access a file larger than 2GB, Oracle fails with the following error:

  ```
  KUP - 04039: unexpected error while trying to find file
  <file name> in director <directory name>
  ```
- Certain commands against the table, such as ANALYZE, will fail.

```
SQL> analyze table products compute statistics;
analyze table products compute statistics

*
ERROR at line 1:
ORA-30657: operation not supported on external organized
Table
```

This limitation is important because most DBAs have scripts that regularly refresh object statistics based on a schema. If you try to generate statistics on an external table, the command will fail.

- The data in external tables isn't backed up as part of regular Oracle backup routines because it's outside the scope of the database.

Performance

One expects the Oracle kernel to incur more overhead when processing external tables. An Oracle TYPE and TYPE BODY named *sys.oracle_loader* exist in the database and process all statements accessing external tables. This process increases the overhead to access the data, and when compared to a regular table is many times slower. Oracle must fetch and perform tasks that it normally doesn't perform (such as conversions, handling rejections, and logging) and is therefore significantly slower. I experimented with the performance of external tables by creating an internal table with the exact data as the external one:

```
SQL> create table products_internal as select * from
products;

Table created.
```

The table contained 5,292 rows, with the same data as in the spreadsheet. The internal table didn't have any indexes or primary keys defined. Based on the script shown in Listing 2, the internal table was consistently 8-10 times faster to access than the external one. Optimally, external tables should be used as a means to load data into internal tables and shouldn't be queried as an external data source.

Listing 2: *Access to the internal table is significantly faster than to the external table.*

```
set term off
col a new_value start
select dbms_utility.get_time() a from dual;
select count(*) from products_internal where product_no = 12313;
col b new_value stop
select dbms_utility.get_time() b from dual;
col c new_value answer
select (&stop - &start) c from dual;
col d new_value start_ext
select dbms_utility.get_time() d from dual;
select count(*) from products where product_no = 12313;
col e new_value stop_ext
select dbms_utility.get_time() e from dual;
col f new_value answer_ext
select (&stop_ext - &start_ext) f from dual;
col ans form 999
col ans_ext form 999
set term on
prompt
prompt
select 'Internal Table Execution Time in ms ', &answer ans
from dual;
select 'External Table Execution Time in ms ', &answer_ext ans_ext
from dual;
```

By taking the following actions, you can minimize the overhead used when processing an external table:

- Use the PARALLEL clause when you create the table. This value indicates the number of access drivers that will be started to process the datafiles and will divide the files into portions that can be processed separately.
- Use datatypes in Oracle that will match the physical data attributes, which will eliminate costly data conversion.

- Use fixed values when possible, including:
    - Fixed-width character sets
    - Fixed-length fields
    - Fixed-length records

  The RECORDS FIXED clause is listed under access parameters and requires the definition of fields. In the following example, the data line is 40 bytes long, plus one byte for the new line. The field names must be the same as the column names to which they correspond.

  ```
  RECORDS FIXED 41
  FIELDS
  (
  emp_first_name char(20)
  emp_last_name char(20)
  )
  ```

- Use single-character delimiters, and use the same character sets as used in the database.
- Minimize rejections since Oracle performs more I/O for each one.

Practical Applications

External tables have many different practical applications, which I'll place into two categories: business processing and database administration.

From the business-processing standpoint, external tables serve a vital need in a data-warehousing environment, in which Extract, Transform, and Load processes are common. External tables make it unnecessary for users to create temporary tables during these processes, thereby reducing required space and the risk of failed jobs. External tables can be used instead of temporary tables and utilities like SQL*Loader. They also provide an easy way for companies to load different information sources into Oracle-whether in Excel, ACT!, or Access, information can be loaded and processed.

From the database administration view, I'm most interested in features that help me do my job. I want to monitor those files that I look at frequently-alert.log and init.ora-without leaving a SQL> prompt. Then I can use SQL commands to query the file and specify WHERE clauses for more sophisticated processing. An example of creating an external table to point to the alert log is as follows:

```
create directory BDUMP AS 'd:\oracle9i\admin\PROD\bdump';
create table alert_log (text varchar2(200))
organization EXTERNAL (
type oracle_loader
default directory BDUMP
access parameters
( records delimited by newline
badfile 'dave.bad'
logfile 'dave.log'
fields terminated by ' '
)
location ('PRODALRT.LOG')
)
reject limit unlimited;
```

Listing 3: *The dba_external_locations view.*

```
SQL> desc dba_external_locations;
Name Null? Type
----------------------------------------- -------- ---------------
OWNER NOT NULL VARCHAR2(30)
TABLE_NAME NOT NULL VARCHAR2(30)
LOCATION VARCHAR2(4000)
DIRECTORY_OWNER CHAR(3)
DIRECTORY_NAME VARCHAR2(30)
```

Database Administration

It's important to know what views in Oracle contain the information pertaining to external tables. The view *dba_tables* shows external tables and has a value of 0 for *pct_free*, *pct_used*, *ini_trans*, and *max_trans*. All other storage columns in the view are null. Scripts that use this view to determine problems should be updated to access *dba_external_tables*. This view contains all of the parameters that you specified when you created the external table.

Another useful view is *dba_external_locations*, which provides a quick way to see which files are accessed from the database (see Listing 3).

Instructors Guide to External Tables

CHAPTER 17

An Oracle Instructor's Guide to Oracle9i - External Tables

This is the second article in a three-part series on Oracle's latest release, Oracle9i. The first article offered information on persistent initialization parameter files, remote startup/shutdown, database managed undo segments, resumable space allocation and flashback query.

In this installment, we'll discuss external tables, tablespace changes, Oracle managed files, multiple blocksizes and cache configuration, on-line table reorganization and index monitoring.

The last article in this series will cover RAC (Real Application Clusters), fail safe, data guard, fine-grained resource management, fine-grained auditing and label security.

External Tables

Seasoned data warehouse administrators know that getting data out of the data warehouse is not the only challenging issue they must address. Extracting, transforming and loading data into the data warehouse can also be quite formidable (and quite expensive) tasks.

Before we begin our discussion on data warehousing, we need to understand that the data warehouse always contains data from external sources. The data is extracted from the source systems, transformed from operational data to business data using business rules, and ultimately, loaded into the data warehouse tables. This process of extracting data from source systems and populating the data warehouse is called Extraction, Transformation and Loading or ETL. Shops deploying data warehouses have the options of purchasing third-party ETL tools or writing scripts and programs to perform the transformation process manually.

Before Oracle9i, the most common methods of manually performing complex transformations were:

- The extracted data would be loaded into staging tables in the data warehouse. The staged data would be transformed in the database and then used as input to programs that updated the permanent data warehouse tables.
- The data would be transformed in flat files stored outside of the database. When the transformation process was complete, the data would be loaded into the data warehouse.

Oracle9i introduces external tables, which provide a mechanism to view data stored in external sources as if it were a table in the database. This ability to read external data provides a more straightforward method of loading and transforming data from external sources. Administrators no longer need to reserve space inside the database for staging tables or write external programs to transform the data outside of the database environment. By making it no longer necessary to stage data in the Oracle database, Oracle9i's external tables have essentially streamlined the ETL function by merging the transformation and loading processes.

External tables in Oracle are read only and cannot have indexes built upon them. Their main use is a data source for more traditional Oracle table structures. Data warehouse administrators are able to use the CREATE TABLE AS SELECT…. and the INSERT INTO…..AS SELECT statements to populate Oracle tables using the external source as input.

Much of the data validation and cleansing that occurs during the ETL process requires access to existing data stored in the data warehouse. Since the external table data is viewed by the database as ordinary table data, SQL, PL/SQL and Java can be used to perform the data transformations. Joins, sorts, referential integrity verification, ID lookups and advanced string manipulations can be performed in the database environment. In addition, advanced SQL statements such as UPSERT and multi-table INSERT statements allow data to be easily integrated into the warehouse environment. The power of the database can be fully utilized to facilitate the transformation process.

External table definitions do not describe how the data is stored externally, rather they describe how the external data is to be presented to the Oracle database engine. Let's take a quick look at an external table definition:

```
CREATE TABLE empxt
(empno      NUMBER(4),
ename       VARCHAR2(10),
job         VARCHAR2(9),
mgr         NUMBER(4),
hiredate    VARCHAR2(20),
sal         NUMBER(7,2),
comm        NUMBER(7,2),
deptno      NUMBER(2))

ORGANIZATION EXTERNAL(TYPE ORACLE_LOADERDEFAULT DIRECTORY dat_dirACCESS
PARAMETERS(records delimited by newlinebadfile
bad_dir:'empxt%a_%p.bad'logfile log_dir:'empxt%a_%p.log'fields
terminated by ','missing field values are null(empno, ename, job,
mgr,hiredate, sal, comm, deptno))LOCATION ('empxt1.dat',
'empxt2.dat'))REJECT LIMIT UNLIMITED;
```

Most of the above table's definition should be familiar to us. However, a few parameters warrant further investigation:

- ORGANIZATION EXTERNAL - Designates that the table's data resides in an external location.
- TYPE - Indicates the access driver. The access driver is the API that interprets the external data for the database. If you do not specify TYPE in the table's definition, Oracle uses the default access driver, *oracle_loader*.
- DEFAULT DIRECTORY - specifies one or more default directory objects that correspond to directories on the file system where the external data resides. Default directories are able to contain both source data and output files (logs, bad files, discard files, etc.). The directory objects that refer to the directories on the file system must already be created with the CREATE DIRECTORY SQL statement. In addition, READ access must be granted to directory objects containing the source data and WRITE access must be granted to all directories that are to contain output files (*bad_dir, log_dir*). Users wanting access to external table data must be granted the appropriate security on the directory objects as well as the table.

- ACCESS PARAMETERS - Assigns values to access driver parameters.
- BADFILE, LOGFILE -Oracle load utility output files.
- LOCATION - Specifies the location for each external data source. The Oracle server does not interpret this clause. The access driver specified interprets this information in the context of the external data.
- PARALLEL (not specified) - Enables parallel query processing on the external data source.

Oracle9i external tables provide great benefits to warehouse environments by combining the transformation and external data access processes. Oracle calls the process "pipelining" and describes it as "a whole new model for loading and transforming external data."

There is a wealth of information available on Oracle9i external tables. Instead of providing you with an in-depth description of how to implement and administer Oracle9i external tables, please refer to Dave Moore's excellent article in DBAzine.com titled "External Tables in Oracle9i." His suggestion to use the external table feature to use SQL statements to search the database alert log is a GREAT idea!

Tablespace Changes

Oracle9i provides the database administrator with a variety (read that bewildering array) of new tablespace parameters and block sizes. Administrators are now able to create Oracle managed tablespaces, user managed tablespaces, locally managed tablespaces, dictionary managed tablespaces, specify

AUTOALLOCATE, UNIFORM, PERMANENT, UNDO as well as select block sizes of 2K, 4K, 8K, 16K, or 32K.

The tablespace definition below combines a few of the aforementioned options:

```
CREATE TABLESPACE oracle_local_auto DATAFILE SIZE 5M BLOCKSIZE 2K;
```

Many of the parameters were not specified intentionally to highlight some of the default specifications for Oracle9i tablespaces. Although some of the parameters we will review were introduced in earlier releases, it is important to discuss them to obtain a clear understanding of tablespace administration in Oracle9i. Let's continue our discussion by taking a closer look at the tablespace's definition:

- Because a datafile specification was not provided, the tablespace is Oracle managed. The datafile clause is only optional if the *db_create_file_dest* initialization parameter is set. The parameter specifies an operating system directory that is the default storage location for Oracle managed datafiles. The operating system directory must already exist and must have the proper security permissions to allow Oracle to create files in it. If a datafile specification and SIZE parameter are not specified, a 100 megabyte file is created by default. During tablespace creation, the database server selects a file name for the Oracle managed file and creates the file in the directory specified in the *db_create_file_dest* initialization parameter. When the tablespace is dropped, Oracle automatically removes the Oracle managed files associated with the dropped tablespace. By default, an Oracle managed datafile is autoextensible with an unlimited maximum size.

- The tablespace will be locally managed because we did not specify EXTENT MANAGEMENT DICTIONARY during creation. Oracle has changed the default from dictionary managed in Oracle8i to locally managed in Oracle9i. Locally managed tablespaces track all extent information in the tablespace itself, using bitmaps. Tracking extents in bitmaps improves speed and concurrency of space operations.

  Administrators are able to override Oracle managed extents by specifying EXTENT MANAGEMENT DICTIONARY in the tablespace definition. Dictionary managed tablespaces rely on data dictionary tables to track space utilization within the tablespace. The SYSTEM tablespace is always dictionary managed.

- The tablespace will use the default free space management setting of SEGMENT SPACE MANAGEMENT MANUAL. As a result, Oracle will use freelists to manage free space within segments in the tablespace. Free lists are lists of data blocks that have space available for inserting rows.

  Administrators have the option of overriding the default specification of SEGMENT SPACE MANAGEMENT MANAUAL with SEGMENT SPACE MANAGEMENT AUTO. SEGMENT SPACE MANAGEMENT AUTO tells Oracle to use bitmaps to manage free space within a segment. The bitmap structure stores information that describes the amount of space in the blocks that are available for row inserts. As free space within each block grows and shrinks, its new state is reflected in the bitmap. Bitmaps allow Oracle to manage free space more automatically. As a result, tracking free space within segments using bitmaps provides a simpler and more efficient method of free space management. Only permanent, locally managed tablespaces can specify automatic segment space management.

- The extent management will be AUTOALLOCATE (extent sizes defined and managed by Oracle) because a default storage clause is not be specified. If the default storage clause is not specified, or if it is specified with PCTINCREASE not equal to 0 and/or INITIAL not equal to NEXT, then Oracle creates a locally managed tablespace with extents managed automatically (AUTOALLOCATE).

  Administrators are also able to specify that the tablespace is managed with uniform extents of a specific size by specifying UNIFORM SIZE in the tablespace's definition or by specifying INITIAL = NEXT and PCTINCREASE = 0. This specification tells Oracle to create a uniform locally managed tablespace with uniform extent size = INITIAL.

- The tablespace datafile will have a 2K blocksize. Oracle9i allows administrators to specify a nonstandard block size for tablespaces. In order for the tablespace specification to override the standard database blocksize specified during database creation, the *db_cache_size* and *db_*n*k_cache_size* (where nk matches the tablespace block size) must be set in the initialization parameter file. Oracle9i allows administrators to choose from 2K, 4K, 8K, 16K and 32K blocksizes.

- Finally, it will be autoextensible (the file will be able to automatically grow in size) because autoextensible is the default for an Oracle managed file.

Let's take a look at our tablespace definition again. This time we will provide all of the specifications for some of the features we have just discussed:

```
CREATE TABLESPACE oracle_local_auto DATAFILE SIZE 5M BLOCKSIZE 2K
AUTOEXTEND ON EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT MANUAL;
```

Oracle9i's new tablespace definitions allow administrators to tailor their environments to meet application requirements. We'll end our discussion on Oracle9i tablespaces with a few quick recommendations:

- Oracle managed - In addition to the benefits of not needing to create filenames and define specific storage requirements, managed files provide the additional advantage of being deleted from the operating system when the DROP TABLESPACE statement is executed. But these benefits do not outweigh the disadvantage of losing the flexibility of specifying different mountpoints or drives manually. Most administrators will prefer to have the flexibility of placing files on different drives or mountpoints and to not be forced into using one directory specification (whether that directory is striped or not).

- Multiple block sizes - Multiple blocksize specifications allow administrators to tailor physical storage specifications to a data object's size and usage to maximize I/O performance. In addition, it also allows administrators to easily use the transportable tablespace feature to transfer tablespaces between databases having different default blocksizes (i.e. moving data from an OLTP application to a data warehouse).

- Locally managed - Oracle is highly recommending that locally managed tablespaces be used for all tablespaces except the SYSTEM tablespace. Because extent management is tracked internally, the need to coalesce tablespaces is no longer required. In addition, allocating or releasing space in a locally managed tablespace avoids recursive space management operations (updates to data dictionary tables that track space utilization). Oracle also states that data objects with high numbers of extents have less of a performance impact on locally managed tablespaces than they do on their dictionary managed counterparts.

Online Table Reorganizations

Oracle9i allows DBAs to perform complex table redefinitions on-line. Administrators now have the capability to change column names and datatypes, manipulate data, add and drop columns and partition tables while the table is being accessed by on-line transactions (for a complete list of changes, please refer to the Oracle9i Administration Guide). This new feature provides significant benefits over more traditional methods of altering tables that require the object to be taken off-line during the redefinition process. Oracle9i provides a set of procedures stored in the PL/SQL package *dbms_redefinition* as the mechanism to perform on-line redefinitions.

Most tables in Oracle can be redefined. The Oracle9i Administration Guide provides a listing of table specifications that will prohibit a table from being redefined on-line. For example, one requirement is that the table being redefined must have a primary key. Oracle9i provides a procedure that will check the table to determine if it can be redefined. The example below shows the table SCOTT.SOURCE_EMP being checked to determine if it meets the on-line redefinition criteria:

```
EXEC dbms_redefinition.can_redef_table ('SCOTT', 'SOURCE_EMP');
```

Administrators create an empty work table in the same schema as the table to be redefined. This work table is created with all of the desired attributes and will become the new table when the redefinition is executed. The two table definitions below show our source table (SCOTT.SOURCE_EMP) and the table containing our desired attributes (SCOTT.WORK_EMP):

```
CREATE TABLE scott.source_emp
(empno NUMBER(4) PRIMARY KEY,
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7, 2),
comm NUMBER(7, 2),
deptno NUMBER(2));

create table scott.work_emp
(enum NUMBER PRIMARY KEY,
lname VARCHAR2(20),
new_col TIMESTAMP,
salary  NUMBER));
```

After the redefinition process is complete, SCOTT.WORK_EMP will become the new SCOTT.SOURCE_EMP table and SCOTT.SOURCE_EMP will become SCOTT.WORK_EMP. The tables are in effect "swapped" during the final phase of transformation.

The next step is to transfer the data from the SCOTT.SOURCE_EMP table to SCOTT.WORK_EMP using the *dbms_redefinition.start_redef_table* procedure. The step also links the two tables together for the remainder of the redefinition process. Administrators code column mappings and data modifications during this step to transform the data. The statement below shows the SCOTT.SOURCE_EMP data being manipulated as it is being transferred to the SCOTT.WORK_EMP table:

```
EXEC dbms_redefinition.start_redef_table
('SCOTT', 'SOURCE_EMP', 'WORK_EMP', 'EMPNO ENUM, ENAM LNAME, SAL*3
SALARY')
```

The above redefinition statement multiplies the SALARY column by three and renames columns EMPNO to ENUM and ENAM to LNAM. The work table also has a new column added (NEW_COL) and does not have column definitions for JOB, MGR, HIREDATE, COMM, DEPTNO.

Triggers, indexes, constraints and grants can now be created on the work table. Referential constraints must be created using the DISABLE option. All triggers, indexes, constraints and grants replace those on the source table being redefined.

The final step of the redefinition process is to execute *dbms_redefinition.finish_redef_table*, which performs the following functions:

- The work table becomes the new source table. The new source table's definition includes all grants, indexes, constraints and triggers created on the work table during the transformation process.
- All referential integrity constraints created on the work table are enabled.
- The source table becomes the new work table. All grants, indexes, constraints and triggers that were on the old source table are also transferred. Referential integrity constraints on the new work table are disabled.
- All DML statements applied to the old source table during the redefinition process are transferred to the work (new source) table.
- The tables are locked for the length of time it takes to perform the table name "swap."

- PL/SQL procedures that access the table being redefined are invalidated. They may remain invalidated if the redefinition process has changed the table structure in such a way that they can no longer successfully access the table data.

During the time period between the executions of *start_redef_table* and *finish_redef_table*, Oracle9i saves all DML changes being applied to the source table. These recorded changes are applied to the work table during the final step of the transformation process. The number of stored changes that need to be applied has a direct affect on the length of time it takes *finish_redef_table* to execute. A large number of changes being applied to the source table during the redefinition process may cause the *finish_redef_table* step to become quite "lengthy." Administrators are able to execute the *dbms_redefinition.sync_interim_table* procedure to periodically synchronize the source and work tables during the period between *start_redef_table* and *finish_redef_table*. Periodically synchronizing the tables reduces the number of stored changes that need to be applied to the work table and the amount of time it takes *finish_redef* to execute.

Oracle9i supplies *dbms_redefinition.abort_redef_table* that can be used to cancel the redefinition process. Administrators are able to abort the process at any time between the executions of *start_redef_table* and *finish_redef_table*.

Index Monitoring

Determining if an index will increase performance is a pretty straightforward process. The administrator is focusing their tuning efforts on a particular table or query and is able to gather the specific information necessary to assist in the decision making process.

Dropping unused indexes is also an important part of application tuning. Indexes force Oracle to occur additional I/O every time a row is inserted or deleted into the table they are built upon. Every update of the table's columns incurs additional I/O to all indexes defined on those columns. Unused indexes also waste space and add unnecessary administrative complexity.

Determining if indexes were being used in releases prior to Oracle9i was a time consuming and error-prone process. EXPLAIN plan and trace output could be used but there was no single mechanism that monitored index usage at the database level.

Oracle9i simplifies the index usage monitoring process by providing the ALTER INDEX……… MONITOR USAGE command. To successfully start or stop a monitoring session, the user must be logged on as the schema owner of the index. The statement below turns monitoring on for the index SCOTT.EMPIDX while the second statement ends the monitoring session:

```
ALTER INDEX scott.empidx MONITORING USAGE;
ALTER INDEX scott.empidx NOMONITORING USAGE;
```

The *v$object_usage* table can then be accessed to determine if the index was used during the monitoring session. When the session is started, Oracle clears the information in *v$object_usage* for the index being monitored and enters a new start time identifying when the index monitoring session started. After the index monitoring session is concluded, the USED column in the *v$object_usage* table will contain the value 'YES' if the index was used during the monitoring session and the value 'NO' if it was not.

In the last installment of this series, we'll discuss RAC (Real Application Clusters), fail safe, data guard, fine-grained resource management, fine-grained auditing and label security.

Thanks and see you in class!

Using Locally-Managed Indexes

CHAPTER 18

Locally Managed Indexes

OK, I'll say it. Oracle does not always work the way I want it to work. The most obvious example of this is how indexes are managed. As data is manipulated, it is evident that the index does not reuse space that it had. For example, if I have a column containing the values A,D,B,E,C,F, and I put an index on this, then the index is created in the following order:

```
A,B,C,D,E,F.
```

This is part of what makes the index access so fast. So when I perform an update and change C to G, I will have the following:

```
A,B, ,D,E,F,G
```

The space in which the C was held is not reused. This actually is a good idea since it makes the update statement much faster than if a complete index rebuild was necessary for every update. The cost for this speed is empty holes in the index. Over time, it becomes evident that the index on the same number of rows slowly takes more space. To get this empty space back, you need to periodically rebuild an index.

Rebuild in the same Tablespace

When you rebuild an index, you have the choice of rebuilding it in the same tablespace or not. Remember that the current index exists until the new one is successfully created. This can give lead to fragmentation in the current tablespace that only worsens over time. An example of this is an index that was initially 256K with a next extent of 64K. If this index had been spread out to 3 extents, you could have the following:

```
Ext1(128k),other index,ext2(64k),other index, ext3(64k), other index
```

If you leave the index definition as it is, the rebuild will recreate the index in the first block that can hold 128k, resulting in:

```
Ext1(128k),other index,ext2(64k),other index, ext(64)3, other
index,temporary(128k)
```

and then:

```
free(128k),other index,free(64k),other index, free(64)3, other
index,ext1(128k)
```

Now there is more free space mixed in with the indexes and if the index grows and can't fit in 128k anymore, you may end up with chunks of free space that are unusable.

No Fragment

To avoid this fragmentation, the common approach is to rebuild all of the indexes in the tablespace into another tablespace, coalesce this tablespace, and then rebuild them back. This means rebuilding the index twice when you want to do it once.

The other option is simply to drop the indexes, coalesce the tablespace, then recreate. This will set any objects depending on this table to an invalid state and they will need to be recompiled. Depending on sizes, it is usually faster to rebuild.

8.1 to the Rescue

To avoid spending the time to rebuild, you should ensure that all extents in the tablespace are the same. Initial is the same as next and all indexes have the same ...WHAT?. Then it doesn't matter if the tablespace becomes fragmented because all the space remains usable. If you are going to do this, you should also take advantage of the new locally managed tablespaces that Oracle provides in V.8.1.

First, create a tablespace and give it a uniform extent size:

```
Create tablespace local64k_idx
Datafile '…/local64k_idx01.dbf' size 512M
Autoextend on next 10M maxsize unlimited
Extent management local uniform size 64k;
```

Next, put the indexes in this tablespace and don't worry about fragmentation.

Now before you start thinking, 'finally, this guy wrote a short article,' here's another important question: When should you decide to rebuild an index and reclaim the empty space within it? I usually say that an index that is in more than 4 extents should be rebuilt. And what if the index is really 1M? Should you rebuild it each time when a rebuild is not needed at all?

More Than One

You probably already know the answer to that question. You will simply have multiple tablespaces, each locally managed at different sizes. Since my tolerance is an index in 4 extents, I create one tablespace at an extent size of 2 blocks, one at 8, one at 32, and one at 128. See how this all falls into my 4's? If I have an 8k block size, then I create a 16k, 64k, 256k, 1M.

So where do you put what? Of course, you have to start with a guess. Go ahead and put them in whichever of the 4 extents you think is correct and analyze all of them. The rebuild script will put each where it belongs.

What Goes Where

The idea of the rebuild is that any index that is between the extent size for this tablespace and the extent size for the next tablespace belongs in this tablespace. You should pull all of these indexes into this tablespace. So we have the following:

TABLESPACE EXTENT	INDEX SIZE
16K	indexes less than 64k
64K	indexes >= 64k and less than 256k
256K	indexes >=256k and less than 1M
1M	indexes >=1M

Break Points

So as not to be fooled by over-allocated indexes, you should check the *leaf_blocks* for the index instead of the bytes. This gives a true picture of space used instead of space allocated.

Assuming you have a block size of 8k, you should first find the number of blocks in 64k to use as your comparison point.

```
variable limit number
begin
select 65536/value into :limit
from v$parameter where name = 'db_block_size';
end;
/
print :limit
```

Script

Each tablespace will have it's own script, but they are all basically the same, as indicated by the following:

```
spool rebuild_local16.sql
select 'alter index '||owner||'.'||index_name||' rebuild' ||chr(10)
'tablespace local16k_idx'||
' nologging;'||chr(10)||
'analyze index '||owner||'.'||index_name||' compute statistics;'from
dba_indexes
where leaf_blocks < :limit
and owner not in ('SYS','SYSTEM')
and last_analyzed is not null
and partitioned= 'NO'
and tablespace_name != 'LOCAL16K_IDX';
spool off
@rebuild_local16.sql
```

For the other tablespaces, use the following `where` clauses:

```
64k: where leaf_blocks >= (:limit) and leaf_blocks < (4*:limit)
256k: where leaf_blocks >= (4*:limit) and leaf_blocks < (16*:limit)
1M: where leaf_blocks >= (16*:limit)
```

See the pattern?

Each tablespace will pull in all the indexes that belong in it. If you have partitioned indexes, just throw in a union with *dba_ind_partitions*.

Note that you are only analyzing indexes when you rebuild them. This entire approach depends on the index being analyzed the first time it is built so you have data to work with.

Conclusion

Last month we talked about how to partition indexes when they get too big. You will see that indexes that are less than a level of 3 do not usually become bigger than 4M. If you do have indexes larger than 4M, you might also want to make a local 4m tablespace.

Now you can rebuild just the indexes that have either spread out or truly grown, without having to worry about fragmentation in these tablespaces. What a relief!

CHAPTER 19

Sizing Oracle Index Segments – Part 1

How Big Should This Index Be?

A client lets us know that they need an index added to the charge back table. Performance is terrible on the new application, and they forgot the index for the file date and status fields. By the way, there are 1,469,176 rows currently in the table.

Of course, we have all heard this before. Moreover, the lack of planning on their part constitutes an emergency for us. We need to get that on there quickly. So how big do we make the index?

In the past, I have gone out and created the index just based on a guess. I would give the initial extent of 10M and the next of 1M and max extents unlimited. The index would then take as many extents as needed. This can be less efficient when I am doing any index scans since I don't necessarily have the extents physically next to each other. I can also overshoot the real size by a large margin.

B-tree Theory

Ideally, the index is in one extent. Let's ignore partitioning for now and just make one index on the complete table. In fact, this came up for one of my clients that is currently on 7.3.3 Oracle, so I do not have the partitioned index option.

Assuming that the index holds the key value and then the address of the row in the table with this value. Then each row in the table must have an entry in the leaf blocks of the index. The branch blocks are used as an index to the leaf nodes. So each block for the leaf nodes needs to be addressed by the next level up of branch nodes.

A (very) simple example:

We are indexing a table that has a single column, containing the letters of the alphabet. Each block holds 4 values. In reality, it would contain a value and an address, we will show only the value just to keep the picture simple:

```
Branch Block 21
P11,Z12
Branch Block 11 12
Value D1,H2,L3,P4 T5,X6,Z7
Leaf Block 1 2 3 4 5 6 7
Value ABCD EFGH IJKL MNOP QRST UVWX YZ
```

You can think of the branch addressing as saying 'values less than _'. If we were looking for the address of the row for 'K' we would visit blocks 21, 11,3.

Estimate Leafs

So armed with this theory, I started with the length of each field plus 1 byte per column for the column header to get the row length:

```
select sum(data_length) + 2 row_length
from dba_tab_columns
where column_name in ('CB_FILE_DATE','CB_STATUS')
and table_name = 'CHARGEBACK_DATA';
```

Let's see how many of these rows will fit into a block. For this we need to decide how much free space we will leave when defining this index. For the example, we will use *pctfree* = 10.

```
Select block_size,row_length
,trunc(block_size * .9 / row_length) rows_per_block
From
(select sum(data_length) + 2 row_length
from dba_tab_columns
where column_name in ('CB_FILE_DATE','CB_STATUS')
and table_name = 'CHARGEBACK_DATA')
,(select value block_size from v$parameter
where name = 'db_block_size');
BLOCK_SIZE ROW_LENGTH ROWS_PER_BLOCK
---------- ---------- --------------
2048 11 167
```

Now let's find the total number of leaf blocks we should need:

```
Select block_size,row_length,rows_per_block,num_rows
,ceil(num_rows/rows_per_block) num_blocks
from
(Select block_size,row_length
,trunc(block_size * .9 / row_length) rows_per_block
from
(select sum(data_length) + 2 row_length
from dba_tab_columns
where column_name in ('CB_FILE_DATE','CB_STATUS')
and table_name = 'CHARGEBACK_DATA')
,(select value block_size from v$parameter
where name = dB_block_size')
)
,(select num_rows from dba_tables
where table_name = 'CHARGEBACK_DATA');

BLOCK_SIZE ROW_LENGTH ROWS_PER_BLOCK NUM_ROWS NUM_BLOCKS
---------- ---------- -------------- ---------- ----------
2048 11 167 1469176 8798
```

So we know we need around 8,798 (remember, we are estimating) blocks. What do we need for branches?

Estimate Branches

For the branches, we logically need to address each leaf from the next layer up. So looking at it simply, we have 8,798 rows (the number of leaf blocks) that need to be addressed. We saw in the previous section that a block hold 167 rows, so we have 8,798/167 for the number of branch blocks on the first level.

```
select rows_per_block,num_blocks
,ceil(num_blocks/rows_per_block) num_branch_blocks
from
(Select block_size,row_length,rows_per_block,num_rows
,ceil(num_rows/rows_per_block) num_blocks
from
(Select block_size,row_length
,trunc(block_size * .9 / row_length) rows_per_block
From
(select sum(data_length) + 2 row_length
from dba_tab_columns
where column_name in ('CB_FILE_DATE','CB_STATUS')
and table_name = 'CHARGEBACK_DATA')
,(select value block_size from v$parameter
where name = dB_block_size')
)
,(select num_rows from dba_tables where table_name = 'CHARGEBACK_DATA')
);
ROWS_PER_BLOCK NUM_BLOCKS NUM_BRANCH_BLOCKS
-------------- ---------- -----------------
167 8798 53
```

For the next level up, we only need 1 block since there are only need 53 blocks to be addressed. So our total need estimate is:

```
(1 + 53 + 8798) * 2048 bytes = 18,128,896 bytes = 17.2M
```

Making the Index

So now we can make our index with at least an educated size guess. Remember that this is only an estimate, we didn't figure out null columns, block header space and any of the other minor things that change the true size of an index.

You can use this same approach for estimating the size of the next extent if you know how many new rows to expect per week. This sizing does not work for bitmap indexes, just the normal b-tree.

Sizing Oracle Index Segments – Part 2

CHAPTER 20

Is This Index the Right Size?

In the first article of this two-part series, we talked about how to estimate a good starting size for a new index. What about the indexes that already exist? Are they the right size? One of the most common problems we see in an existing system is that the indexes were made based on some estimate and now we need to figure out if they are the right size.

Validate Structure

The first way I learned to check the correct size of an index is using the 'analyze index … validate structure' . This command then puts the results in the *index_stats* table and gives you a detailed view of your index. The problems I started to encounter with this method are that it can be slow; this 'analyze' command locks out users; and the results are different for partitioned and sub-partitioned indexes. So I started looking for a new way. Turns out it was staring me in the face all along.

Dba_Indexes

We are usually analyzing our tables at least weekly, some nightly. So the statistics we need are already available.

As of version 8.0, Oracle added *num_rows* to the *dba_indexes* table. Previously we would have joined *dba_indexes* to *dba_tables* to get *num_rows* for an index. In doing this, however, we don't take into account where the index value might be null and therefore not be included. The number of rows in the index will be less than or equal to the number of rows in the table. So the added field is the final piece of information needed to determine how much space this index really needs.

To see the current space used in blocks for the indexes on the table TASK, we can run the following:

```
select a.owner,a.index_name,a.leaf_blocks,a.num_rows
from dba_indexes a
where a.owner not in ('SYS','SYSTEM')
and a.index_type!='BITMAP'
and a.last_analyzed is not null
and a.table_name = 'TASK';
```

OWNER	INDEX_NAME	LEAF_BLOCKS	NUM_ROWS
ORADBA	TASK_APPT_FK	65	23771
ORADBA	TASK_CENTER_FK	99	45975
ORADBA	TASK_CLIENT_FK	101	45975
ORADBA	TASK_ENC_FK	117	45975
ORADBA	TASK_TYPE_FK	74	45975
ORADBA	TASK_DT_REQ	84	45975

Given this, we see that 23,771 rows of the index *task_appt_fk* currently fit into 65 blocks. There might be unused space in these 65 blocks, but we will deal with that later. This means that we have approximately 366 rows per block (23,771/65). Using the logic presented in the first article on B-tree sizing, if we had more than 366 blocks, then we would have a level of branch blocks. But this index has just the one root block. Our total need for this index is then *65 leaf_blocks + 1 root block = 66 blocks*, as it is currently defined.

So how much space was given to this index? This is an answer we always have in *dba_segments*. Let's add it to our query:

```
select a.owner,a.index_name,a.leaf_blocks,a.num_rows,b.blocks
from dba_indexes a
,dba_segments b
where a.owner not in ('SYS','SYSTEM')
and a.index_type!='BITMAP'
and a.last_analyzed is not null
and a.table_name = 'TASK'
and a.owner = b.owner
and a.index_name = b.segment_name;
```

OWNER	INDEX_NAME	LEAF_BLOCKS	NUM_ROWS
ORADBA	TASK_APPT_FK	65	23771
ORADBA	TASK_CENTER_FK	99	45975
ORADBA	TASK_CLIENT_FK	101	45975
ORADBA	TASK_ENC_FK	117	45975
ORADBA	TASK_TYPE_FK	74	45975
ORADBA	TASK_DT_REQ	84	45975

We can see then that the *task_appt_fk* index needs 66 blocks and currently is holding 175. So we could free up 109 blocks if we rebuilt this index at the correct size. This was all done based on the previously run statistics and without locking out any user. I can do this at any time during the day.

Logical Steps for Resizing and Defragging

So how do we use this newfound power for good? The best case here is that we do a complete tablespace so we can resize and defrag the tablespace at the same time. Here are the logical steps:

- Recreate the indexes at the size based on *dba_indexes* in a different tablespace. This will flush out any deleted space within the index.
- Coalesce the index tablespace
- Analyze these indexes to get up-to-the-minute statistics
- Recreate the indexes back into the index tablespace

We want an easy way to specify the indexes we are working on for each of the steps. I normally just drop the names into a table that will exist only for the duration of this exercise. Be sure to drop that temporary table when you've finished. We also pull out the block size to make the statements faster later on:

```
create table t_names
storage (initial 64K next 64K pctincrease 0)
pctfree 0 pctused 80
as select owner,index_name
from dba_indexes
where tablespace_name = 'HRX';
variable block_size number
begin
select to_number(value) into :block_size
from v$parameter where name = 'db_block_size';
end;
/
```

Now we will spool out the analyze statement for our indexes and run it if we know the statistics are old. This does not lock out users:

```
spool c:\analyze_index.sql
select 'analyze index '||owner||'.'||index_name||' compute statistics;'
from t_names;
spool off
@c:\analyze_index
```

The results are as follows:

```
analyze index ORADBA.TASK_DT_REQ compute statistics;
analyze index ORADBA.TASK_TYPE_FK compute statistics;
analyze index ORADBA.TASK_ENC_FK compute statistics;
analyze index ORADBA.TASK_CLIENT_FK compute statistics;
analyze index ORADBA.TASK_APPT_FK compute statistics;
analyze index ORADBA.TASK_CENTER_FK compute statistics;
```

The next step requires that we give the owners of the indexes we are moving, rights on the new target tablespace. You can check *dba_ts_quotas* and *dba_sys_privs* to see if they already have rights:

```
select distinct 'alter user '||owner||' quota unlimited on hrd;'
from t_names a
where not exists (select 'x' from dba_sys_privs
where a.owner = grantee and rownum =1)
and not exists (select 'x' from dba_ts_quotas
where tablespace_name = 'HRD'
and a.owner = username and rownum =1);
```

Here are the results:

```
alter user ORADBA quota unlimited on hrd;
```

When we have finished, we want to be sure to set the quota back to 0 for all the owners we changed. Now the groundwork is complete, so we can get to the command that will perform the actual move:

```
spool c:\resize_new_index.sql
select 'alter index '||a.owner||'.'||a.index_name||
' rebuild tablespace hrd'||chr(10)||
' storage(initial '||
((decode(blevel
,0,0
,ceil(a.leaf_blocks/trunc(a.num_rows/a.leaf_blocks)))
+a.leaf_blocks)
*:block_size)||
' next '||a.next_extent||
' maxextents '||a.max_extents||' pctincrease 0)'||
' nologging;'
from dba_indexes a
,dba_segments b
,t_names c
where a.leaf_blocks > 0
and a.num_rows > 0
and a.owner not in ('SYS','SYSTEM')
and a.index_type != 'BITMAP'
and a.last_analyzed is not null
and a.owner = b.owner
and a.index_name = b.segment_name
and a.owner = c.owner
and a.index_name = c.index_name
order by (decode(blevel,0,0
,ceil(a.leaf_blocks/trunc(a.num_rows/a.leaf_blocks)))+a.leaf_blocks)
;
spool off
```

And what follows are the results:

```
alter index ORADBA.TASK_APPT_FK rebuild tablespace hrd
storage(initial 1081344 next 2097152 maxextents 1017 pctincrease 0)
nologging;
alter index ORADBA.TASK_TYPE_FK rebuild tablespace hrd
storage(initial 1228800 next 2097152 maxextents 1017 pctincrease 0)
nologging;
alter index ORADBA.TASK_DT_REQ rebuild tablespace hrd
storage(initial 1392640 next 65536 maxextents 1017 pctincrease 0)
nologging;
alter index ORADBA.TASK_CENTER_FK rebuild tablespace hrd
storage(initial 1638400 next 1048576 maxextents 1017 pctincrease 0)
nologging;
alter index ORADBA.TASK_CLIENT_FK rebuild tablespace hrd
storage(initial 1671168 next 2097152 maxextents 1017 pctincrease 0)
nologging;
alter index ORADBA.TASK_ENC_FK rebuild tablespace hrd
storage(initial 1933312 next 2097152 maxextents 1017 pctincrease 0)
nologging;
```

All Together Now

So we move, coalesce, analyze and move back. We have rebuilt the indexes at the correct size and defragged the tablespace. There was never a time when an index did not exist, and there is no risk of dropping one. A word of caution: Users will be impacted during the move, so you want to do this during off-peak hours. But we don't have to lock users out just to figure out what to do.

The initial extent is computed as the number of leaf blocks plus the number of computed branch blocks, multiplied by the block size. Notice that I ordered this by the size of the index. This is based on the assumption that the larger indexes usually have more size activity; it's also an effort to minimize future fragmentation.

You will also notice that this ignores the empty indexes, those with 0 rows or 0 blocks. These indexes should also be rebuilt at 1 block for the initial extent or dropped, unless you know this index will be populated in the near future. For partition indexes, use the same query but with *dba_ind_partitions* in place of *dba_indexes*. The partition name must be included in the rebuild statement.

Once this is done and in place, there is no excuse for not knowing how big the index really should be.

Oracle Partitioning Design

CHAPTER 21

Partitioning in Oracle 9i, Release 2

Learn how to use the various partitioning methods in Oracle 9i Release 2.

This is the first part of a two-part article addressing "How To" partition in Oracle 9i, Release 2. Part 1 will cover the basics of partitioning and how to partition tables. Part 2 will cover the partitioning of indexes. Part 2 will also draw together the concepts from the entire article into real life examples.

Introduction

Oracle DBAs face an ever growing and demanding work environment. The only thing that may outpace the demands of the work place is the size of the databases themselves. Database size has grown to a point where they are now measured in the hundreds of gigabytes, and in some cases, several terabytes. The characteristics of very large databases (VLDB) demand a different style of administration. The administration of VLDB often includes the use of partitioning of tables and indexes.

Since partitioning is such an integral part of VLDB the remainder of this article will focus on how to partition, specifically, the partitioning of tables in an Oracle 9i Release 2 environment. Part 2 of this article will focus on the partitioning of indexes. The complete article will cover:

- Partitioning Defined

- When To Partition
- Different Methods Of Partitioning
- Partitioning Of Tables
- Partitioning Of Indexes

The organization of this article is modular so you can skip to a specific topic of interest. Each of the table partitioning methods (Range, Hash, List, Range-Hash and Range-List) will have its own section that includes code examples and check scripts.

Background

This article assumes that Oracle 9i Release 2 is properly installed and running. You will also need to have a user account that has a minimum of `Cre`ate Table, Alter Table and Drop Table privileges. In addition to the basic privileges listed above, the creation of five small tablespaces (TS01, TS02, TS03, TS04, TS05) or changes to the tablespace clause will need to be done to use the examples provided in this article.

Ideally, you should try each of the scripts in this article under a DBA role. All scripts have been tested on Oracle 9i Release 2 (9.2) running on Windows 2000.

Partitioning Defined

The concept of divide and conquer has been around since the times of Sun Tzu (500 B.C.). Recognizing the wisdom of this concept, Oracle applied it to the management of large tables and indexes. Oracle has continued to evolve and refine its partitioning capabilities since its first implementation of range partitioning in Oracle 8. In Oracle 8i and 9i, Oracle has continued to add both functionality and new partitioning methods. The current version of Oracle 9i Release 2 continues this tradition by adding new functionality for list partitioning and the new range-list partitioning method.

When To Partition

There are two main reasons to use partitioning in a VLDB environment. These reasons are related to management and performance improvement.

Partitioning offers:

- Management at the individual partition level for data loads, index creation and rebuilding, and backup/recovery. This can result in less down time because only individual partitions being actively managed are unavailable.
- Increased query performance by selecting only from the relevant partitions. This weeding out process eliminates the partitions that do not contain the data needed by the query through a technique called partition pruning.

The decision about exactly when to use partitioning is rather subjective. Some general guidelines that Oracle and I suggest are listed below.

Use partitioning:

- When a table reaches a "large" size. Large being defined relative to your environment. Tables greater than 2GB should always be considered for partitioning.
- When performance benefits outweigh the additional management issues related to partitioning.

- When the archiving of data is on a schedule and is repetitive. For instance, data warehouses usually hold data for a specific amount of time (rolling window). Old data is then rolled off to be archived.

Take a moment and evaluate the criteria above to make sure that partitioning is advantageous for your environment. In larger environments partitioning is worth the time to investigate and implement.

Different Methods of Partitioning

Oracle 9i, Release 2 has five partitioning methods for tables. They are listed in the table below with a brief description.

PARTITIONING METHOD	BRIEF DESCRIPTION
Range Partitioning	Used when there are logical ranges of data. Possible usage: dates, part numbers, and serial numbers.
Hash Partitioning	Used to spread data evenly over partitions. Possible usage: data has no logical groupings.
List Partitioning	Used to list together unrelated data into partitions. Possible usage: a number of states list partitioned into a region.
Composite Range-Hash Partitioning	Used to range partition first, then spreads data into hash partitions. Possible usage: range partition by date of birth then hash partition by name; store the results into the hash partitions.
Composite Range-List Partitioning	Used to range partition first, then spreads data into list partitions. Possible usage: range partition by date of birth then list partition by state, then store the results into the list partitions.
Range Partitioning	Used when there are logical ranges of data. Possible usage: dates, part numbers, and serial numbers.

For partitioning of indexes, there are global and local indexes. Global indexes provide greater flexibility by allowing indexes to be independent of the partition method used on the table. This allows for the global index to reference different partitions of a single table. Local indexes (while less flexible than global) are easier to manage. Local indexes are mapped to a specific partition. This one-to-one relationship between local index partitions and table partitions allows Oracle the ability to manage local indexes. Partitioning of indexes will be the focus of Part 2 of this article.

Detailed examples and code will be provided for each partitioning method in their respective sections. The use of the ENABLE ROW MOVEMENT clause is included in all of the examples of table partitioning to allow row movement if the partition key is updated.

Partitioning Of Tables

Range Partitioning

Range partitioning was the first partitioning method supported by Oracle in Oracle 8. Range partitioning was probably the first partition method because data normally has some sort of logical range. For example, business transactions can be partitioned by various versions of date (start date, transaction date, close date, or date of payment). Range partitioning can also be performed on part numbers, serial numbers or any other ranges that can be discovered.

The example provided for range partition will be on a table named *partition_by_range* (what else would I call it?). The *partition_by_range* table holds records that contain the simple personnel data of FIRST_NAME, MIDDLE_INIT, LAST_NAME, BIRTH_MM, BIRTH_DD, and BIRTH_YYYY. The actual partitioning is on the following columns BIRTH_YYYY, BIRTH_MM, and BIRTH_DD. The complete DDL for the PARTITION_BY_RANGE table is provided in the script *range_me.sql*.

A brief explanation of the code follows. Each partition is assigned to its own tablespace. The last partition is the "catch all" partition. By using *maxvalue* the last partition will contain all the records with values over the second to last partition.

Hash Partitioning

Oracle's hash partitioning distributes data by applying a proprietary hashing algorithm to the partition key and then assigning the data to the appropriate partition. By using hash partitioning, DBAs can partition data that may not have any logical ranges. Also, DBAs do not have to know anything about the actual data itself. Oracle handles all of the distribution of data once the partition key is identified.

The *hash_me.sql* script is an example of a hash partition table. Please note that the data may not appear to be distributed evenly because of the limited number of inserts applied to the table.

A brief explanation of the code follows. The PARTITION BY HASH line is where the partition key is identified. In this example the partition key is AGE. Once the hashing algorithm is applied each record is distributed to a partition. Each partition is specifically assigned to its own tablespace.

List Partitioning

List partitioning was added as a partitioning method in Oracle 9i, Release 1. List partitioning allows for partitions to reflect real-world groupings (e.g.. business units and territory regions). List partitioning differs from range partition in that the groupings in list partitioning are not side-by-side or in a logical range. List partitioning gives the DBA the ability to group together seemingly unrelated data into a specific partition.

The *list_me.sql* script provides an example of a list partition table. Note the last partition with the DEFAULT value. This `DEFAULT` value is new in Oracle 9i, Release 2.

A brief explanation of the code follows. The PARTITION BY LIST line is where the partition key is identified. In this example, the partition key is STATE. Each partition is explicitly named, contains a specific grouping of VALUES and is contained in its own tablespace. The last partition with the DEFAULT is the "catch all" partition. This catch all partition should be queried periodically to make sure that proper data is being entered.

Composite Range-Hash Partitioning

Composite range-hash partitioning combines both the ease of range partitioning and the benefits of hashing for data placement, striping, and parallelism. Range-hash partitioning is slightly harder to implement. But, with the example provided and a detailed explanation of the code one can easily learn how to use this powerful partitioning method.

The *range_hash_me.sql* script provides an example of a composite range-hash partition table.

A brief explanation of the code follows. The PARTITION BY RANGE clause is where we shall begin. The partition key is (BIRTH_YYYY, BIRTH_MM, BIRTH_DD) for the partition. Next, the SUBPARTITION BY HASH clause indicates what the partition key is for the subpartition (in this case FIRST_NAME, MIDDLE_INIT, LAST_NAME). A SUBPARTITION TEMPLATE then defines the subpartition names and their respective tablespace. Subpartitions are automatically named by Oracle by concatenating the partition name, an underscore, and the subpartition name from the template. Remember that the total length of the subpartition name should not be longer than thirty characters including the underscore.

I suggest that, when you actually try to build a range-hash partition table, you do it in the following steps:

1. Determine the partition key for the range.
2. Design a range partition table.
3. Determine the partition key for the hash.
4. Create the SUBPARTITION BY HASH clause.
5. Create the SUBPARTITION TEMPLATE.

Do Steps 1 and 2 first. Then you can insert the code created in Steps 3 -5 in the range partition table syntax.

Composite Range-List Partitioning

Composite range-list partitioning combines both the ease of range partitioning and the benefits of list partitioning at the subpartition level. Like range-hash partitioning, range-list partitioning needs to be carefully designed. The time used to properly design a range-list partition table pays off during the actual creation of the table.

The *range_list_me.sql* script provides an example of a composite range-list partition table.

A brief explanation of the code follows. The PARTITION BY RANGE clause identifies the partition key (BIRTH_YYYY, BIRTH_MM, BIRTH_DD). A SUBPARTITION TEMPLATE then defines the subpartition names and their respective tablespace. Subpartitions are automatically named by Oracle by concatenating the partition name, an underscore, and the subpartition name from the template. Remember that the total length of the subpartition name should not be longer than thirty characters including the underscore.

When building a range-list partition table you may want to refer to the steps mentioned at the end of the Composite Range-List section. The only difference is in Step 4. Instead of "Create the SUBPARTITION BY HASH clause" it would read, "Create the SUBPARTITION BY LIST clause" for the range-list partition table.

Conclusion

This is the first of a two-part article suggesting the use of partition tables in VLDB environments. Part two of this article will cover partition indexes. In part two both methods (partition tables and indexes) will be brought together in real life examples. Look for part two next month.

CHAPTER 22

Oracle Partitioning Design – Part 2

Partitioning in Oracle 9i, Release 2 -- Part 2

Learn how to use the various partitioning methods in Oracle 9i Release 2.

This is the second part of a two-part article addressing "How To" partition in Oracle 9i Release 2. Part 1 covers the basics of partitioning and how to partition tables. Part 2 will cover the partitioning of indexes. Part 2 will also draw together the concepts from the entire article into real life examples.

Introduction

In Part 1 of "Partitioning in Oracle 9i Release 2," we learned how to use the various table partitioning methods in the latest release of Oracle. We will now continue on and learn about Globally Partitioned and Locally Partitioned Indexes. We will cover:

- Background/Overview
- Globally Partitioned Indexes
- Locally Partitioned Indexes
- When To Use Which Partitioning Method
- Real-Life Example

Background

This article assumes that Oracle 9i Release 2 is properly installed and running. You will also need to have a user account that has a minimum of Create Index, Alter Index and Drop Index privileges. In addition to the basic privileges listed above, the creation of five small tablespaces (ITS01, ITS02, ITS03, ITS04, ITS05) or changes to the tablespace clause will need to be done to use the examples provided in this article.

Ideally, you should try each of the scripts in this article under a DBA role. All scripts have been tested on Oracle 9i Release 2 (9.2) running on Windows 2000. The examples below build off of the examples that were used in Part 1 of this article.

Globally Partitioned Indexes

There are two types of global indexes, non-partitioned and partitioned. Global non-partitioned indexes are those that are commonly used in OLTP databases (refer to Figure1). The syntax for a globally non-partitioned index is the exactly same syntax used for a "regular" index on a non-partitioned table. Refer to *gnpi_me.sql* (http://www.dbazine.com/code/GNPI_ME.SQL) for an example of a global non-partitioned index.

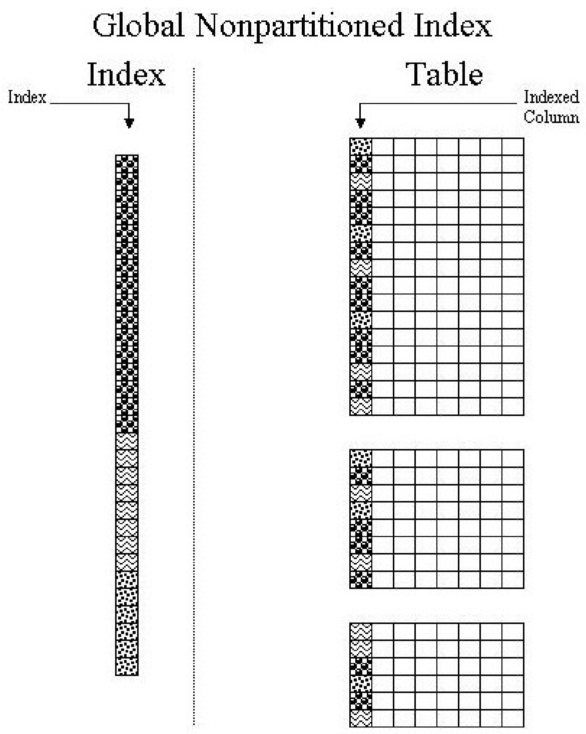


Figure 1

The other type of global index is the one that is partitioned. Globally partitioned indexes at this time can only be ranged partitioned and has similar syntactical structure to that of a range-partitioned table. *gpi_me.sql* (http://www.dbazine.com/code/GPI_ME.SQL) is provides for an example of a globally partitioned index. Note that a globally partitioned index can be applied to any type of partitioned table. Each partition of the globally partitioned index can and may refer to one or more partitions at the table level. For a visual representation of a global partitioned index refer to Figure 2.

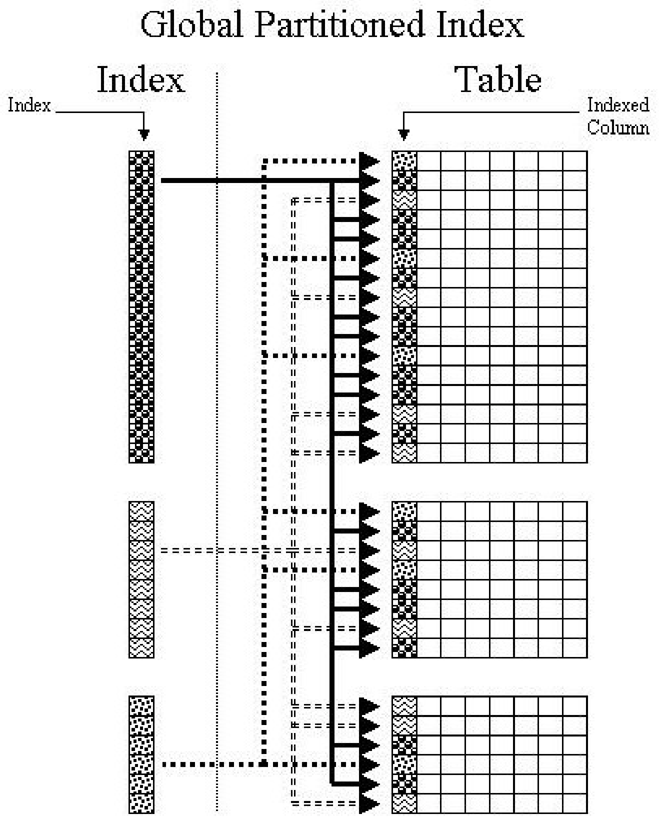


Figure 2

The maintenance on globally partitioned indexes is a little bit more involved compared to the maintenance on locally partitioned indexes. Global indexes need to be rebuilt when there is DDL activity on the underlying table. The reason why they must be rebuilt is that DDL activity often causes the global indexes to be usually marked as UNUSABLE. To correct this problem there are two options to choose from:

- Use ALTER INDEX <*index_name*> REBUILD;

- Or use UPDATE GLOBAL INDEX clause when using ALTER TABLE.

The syntax for the ALTER INDEX statement is relatively straightforward so we will only focus on the UPDATE GLOBAL INDEX clause of the ALTER TABLE statement. The UPDATE GLOBAL INDEX is between the partition specification and the parallel clause. The partition specification can be any of the following:

- ADD PARTITION | SUBPARTITION (hash only)
- COALESCE PARTITION | SUBPARTITION
- DROP PARTITION
- EXCHANGE PARTITION | SUBPARTITION
- MERGE PARTITION
- MOVE PARTITION | SUBPARTITION
- SPLIT PARTITION
- TUNCATE PARTITION | SUBPARTITION

For example:

```
ALTER TABLE <TABLE_NAME>
<PARTITION SPECIFICATION>
UPDATE GLOBAL INDEX
PARALLEL (DEGREE #)
```

Locally Partitioned Indexes

Locally partitioned indexes are for the most part very straightforward. The *lpi_me.sql* (http://www.dbazine.com/code/LPI_ME.SQL) script shows examples of this type of index. In the script, locally partitioned indexes are created on three differently partitioned tables (range, hash, and list). Figure 3 gives a visual representation of how a locally partitioned index works.

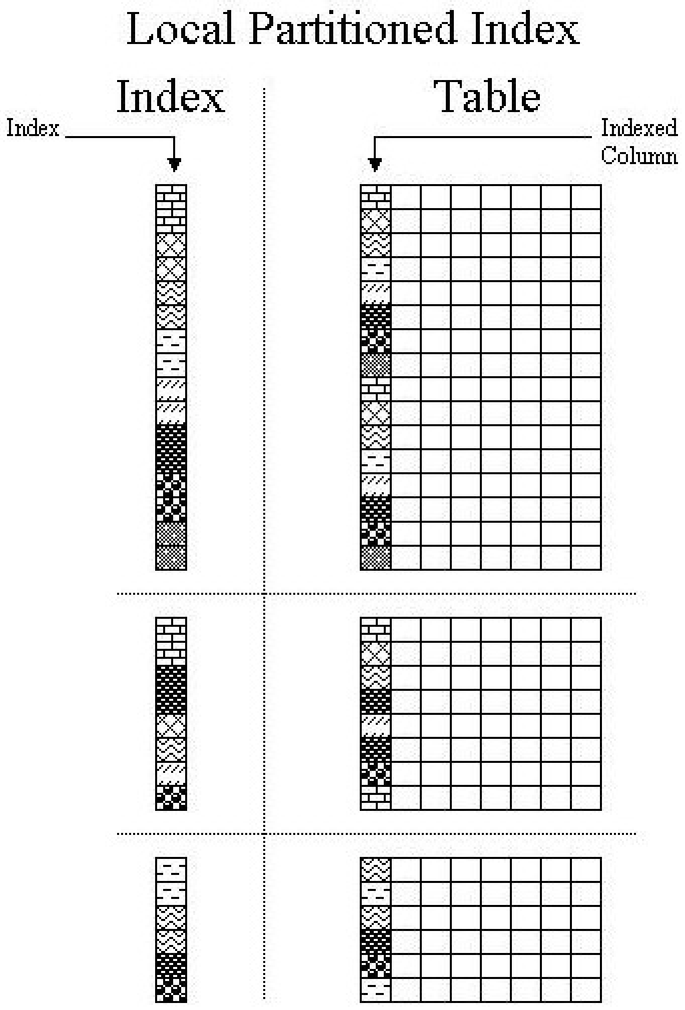


Figure 3

Extra time should be allocated when creating locally partitioned indexes on range-hash or range-list partitioned tables. There are a couple reasons that extra time is needed for this type of index. One of the reasons is a decision needs to be made on what the index will be referencing in regards to a range-hash or range-list partitioned tables. A locally partitioned index can be created to point to either partition level or subpartition level.

Script *lpi4cpt1_me.sql* (http://www.dbazine.com/code/LPI4CPT1_ME.SQL) is the example for the creation of two locally partitioned indexes. This scripts show how to create a locally partitioned index on both a range-hash and range-list partitioned tables at the partition level. Each of the partitions of the locally partitioned indexes is assigned to its own tablespace for improved performance.

When creating a locally partitioned index one needs to keep in mind the number of subpartitions of the range-hash or range-list partitioned table being indexed. Reason being, is that the locally partitioned index will need to reference each subpartition of the range-hash or range-list partitioned table. So, for the locally partitioned index created by *lpi4cpt2.me.sql* (http://www.dbazine.com/code/LPI4CPT2_ME.SQL), this means that one index references twenty-five different subpartitions. For a visual representation of this refer to Figure 4. Script *lpi4cpt3_me.sql* (http://www.dbazine.com/code/LPI4CPT3_ME.SQL) is provided as an example of locally partitioned index on a range-list partition table.

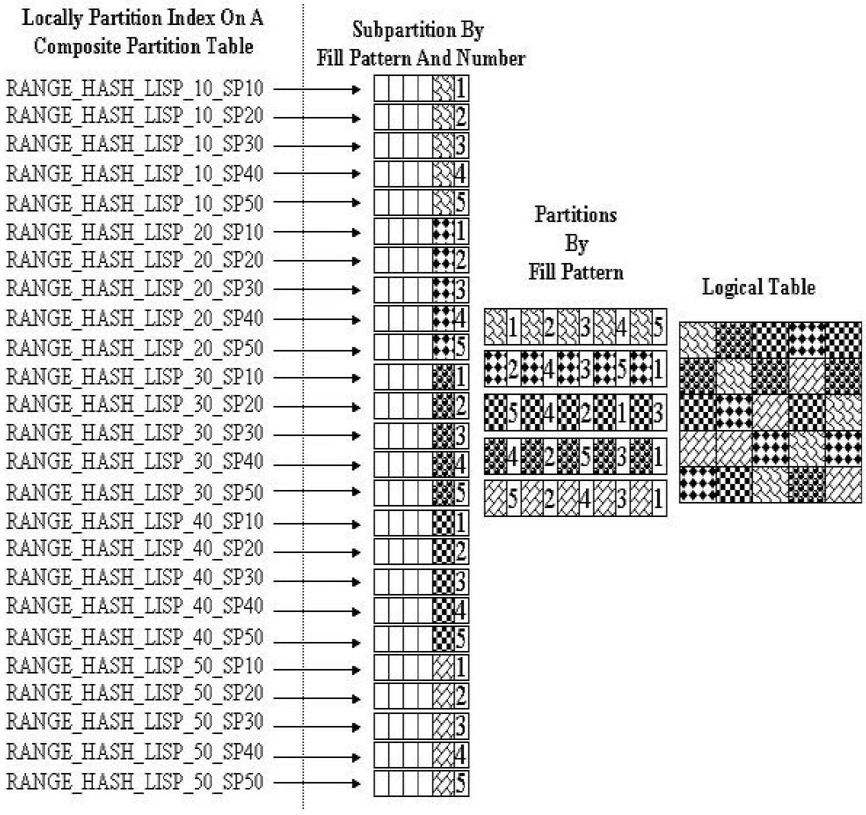


Figure 4

Note: At this time Oracle has not implemented a SUBPARTITION TEMPLATE clause for the creation of locally partitioned indexes on range-hash or range-list partition tables. This means that you need to type everything out as in the examples in *lpi4cpt2_me.sql* and *lpi4cpt3_me.sql.*

Maintenance of locally partitioned indexes is much easier than the maintenance of globally partitioned indexes. Whenever there is DDL activity on the underlying indexed table Oracle rebuilds the locally partitioned index.

This automatic rebuilding of locally partitioned indexes is one reason why most DBAs prefer locally partitioned indexes.

When to Use Which Partitioning Method

There are five different table partitioning methods (range, hash, list, range-hash and range-list) and three for indexes (global non-partitioned, global partitioned and locally partitioned). So, the obvious question is: "When do I use which combination of table and index partitioning?" There is no concrete answer for that question. However, here are some general guidelines on mixing and matching table and index partitioning.

- First determine if you need to partition the table.
    - Refer to Part 1 of this article under "When To Partition"
- Next decide which table partitioning method is right for your situation.
    - Each method is described in Part 1 of this article under "Different Methods of Partitioning"
- Determine how volatile the data is.
    - How often are there inserts, updates and deletes?
- Choose your indexing strategy: global or local partitioned indexes.
    - Each type has its own maintenance consideration.

These guidelines are good place to start when developing a partitioning solution.

Real Life Example

The "rolling window" concept of only retaining a certain amount of data is the norm in most data warehousing environments. This rolling window can also used to archive data from an OLTP system. For our example we will assume that there is a twelve month rolling window.

Our example will cover the following steps:

- Create a range partition table that has a locally partitioned index.
- Use "CREATE TABLE . . AS" to copy the data into a separate table.
- Archive off the table created to hold the rolled off data.
- Drop last month partition.
- Add new months partition.

Script *example.sql* is an annotated code of the example above.

Conclusion

During the course of this two part article we have covered the "How to" of partitioning in Oracle 9i Release 2. Part 1 covered the basics of table partitioning. Part 2 followed with partitioning of indexes. We then brought together both partitioning methods and evaluated when to use each method. Near the end of this article we applied what we have learned in a real life example. I hope that by reading this that this article give you the basic knowledge to evaluate and use partitioning in your next design and implementation of Oracle 9i Release 2.

Effective Segment Partitioning – Part 1

CHAPTER 23

Perils and Pitfalls in Partitioning — Part 1

Partitioning is a favorite topic for authors, presenters, and general DBA community, but most of the papers dwell on the basics and fundamental concepts behind partitioning. The invariable action of most DBAs, after learning the ropes, is to jump into their databases with partitioning in mind. This article describes some of the potential problems — little or non-documented features that may create unanticipated (and unwanted) situations to which you should be alert, and how to resolve them. Understanding these potential problems will go a long way in designing a proper partitioning scheme for your database. Caution: to get the most from this article, you should already have basic knowledge about partitioning; this article is not a primer on that subject.

Plan Table Revisited

Before we begin, let's touch upon a very familiar table for identifying query execution paths that's been available for a long time — the *plan_table*. You've certainly been using this table already, to identify the optimizer plan of a statement. We will examine three specific columns in this table (four, in Oracle9i) that are important for the partitioning option. Here is a basic explanation of these columns.

PARTITION_START	When the optimizer searches a range of partitions for data, this column indicates the PARTITION_ID of the starting partition in that range.
PARTITION_STOP	When the optimizer searches a range of partitions, this column indicates the PARTITION_ID of the last partition of the range.
PARTITION_ID	Each step in the optimizer plan is identified by a unique number called STEP_ID. This column displays the STEP_ID of the step in PLAN_TABLE that decided the beginning and ending PARTITION_IDs.
FILTER_PREDICATES	The exact condition used to evaluate and arrive at the start and stop PARTITION_IDs (9i only).

More information and explanation about these columns will be provided later in the document along with examples.

The New Tool DBMS_XPLAN

It might be useful to describe an exciting tool available in 9i, a new package called *dbms_xplan*, which is useful for querying the *plan_table* data. Instead of writing a complicated SQL statement to see the optimizer plan from the *plan_table*, a call to the *dbms_xplan* displays the optimizer plan in a formatted fashion, making it easier to use. To select the optimizer plan for the last "explain plan" statement, simply use the query,

```
select * from table(dbms_xplan.display(format=>'BASIC'))
```

Using the operator TABLE() (or, performing a CAST operation) makes the return values from the function behave just like rows in a table so they can be queried as if being selected from a relational table.

```
 PLAN_TABLE_OUTPUT
----------------------------------------------

--------------------------------------------
| Id  | Operation             |  Name       |
--------------------------------------------
|   0 | SELECT STATEMENT      |             |
|   1 |  SORT AGGREGATE       |             |
|   2 |   NESTED LOOPS        |             |
|   3 |    TABLE ACCESS FULL  | PTEST3HA    |
|   4 |    TABLE ACCESS FULL  | PTEST3HB    |
--------------------------------------------
```

The display() function takes three arguments:

TABLE_NAME	The name of the table in which the optimization plan is stored; defaults to PLAN_TABLE.
STATEMENT_ID	The statement ID from the plan table mentioned earlier. By default, it takes the last ID, or NULL.
FORMAT	This controls the way the display is formatted (explained later in detail).

Let's examine the last parameter, FORMAT, which is used to control how the output is displayed. It accepts four values as follows:

BASIC	It provides only the minimum amount of information, as in case of the example above, similar to a query from PLAN_TABLE directly.
TYPICAL	This is the default value. It provides a variety of the information useful for understanding how the optimizer works for this statement. For instance, in case of partitioned table operation, the columns PARTITION_START, PARTITION_STOP, PARTITION_ID, and FILTER_PREDICATES are displayed in addition to COST for that step, the number of rows expected to be retrieved, and number of bytes those rows may have. This provides the information to understand statements involving partitioned objects.
ALL	This setting displays all the information displayed for the BASIC and TYPICAL values, and also displays parallel query operations and the related SQL statements, if those are involved.

SERIAL	This setting gets results similar to those retrieved by the TYPICAL setting, but the queries are explained serially even if a parallel query will be used.

Needless to say, the BASIC setting does not provide much information pertaining to partitioned objects, so the TYPICAL setting is recommended. However, the BASIC setting also widens the display. Before running the query, you should make the line size 120 or more. Here is the an output from the same query cited above using format=>'TYPICAL' or with no parameters:

```
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------
--


----------------------------------------------------------------------
--
| Id | Operation         | Name   | Rows | Bytes | Cost | Pstart| Pstop
|
----------------------------------------------------------------------
--
| 0  | SELECT STATEMENT  |        | 5    |   575 |    4 |       |
|
|* 1 | TABLE ACCESS FULL | PTEST1 | 5    |   575 |    4 |     2 |     2
|
----------------------------------------------------------------------
--

Predicate Information (identified by operation id):
---------------------------------------------------

1 - filter("PTEST1"."COL1"=1500)

Note: cpu costing is off

14 rows selected.
```

This example shows that the optimizer will search only partitions with PARTITION_IDs from 2 to 2; i.e., it will search only PARTITION_ID 2. The decision to search that partition was made at STEP_ID 1, as displayed under Predicate Information below the formatted output. The result also mentions that the optimizer decided to select the step based on the information provided to it from the query (or, the filter predicate in the query COL1=1500. This kind of information is extremely useful determining optimizer plans for partitioned objects.

Partition Pruning or Elimination

Given this background information, let's jump into our discussion on partitioning mysteries. The main advantage of partitioning comes when the optimizer chooses the data in a specific partition only, where the requested data will be found and not all the partitions. For instance, consider a table, SALES, partitioned on ORDER_DATE, with one partition per quarter. When the following query is issued,

```
SELECT … FROM SALES
WHERE ORDER_DATE = '1/1/2003'
```

the optimizer does not go through the entire table, but only the partition that houses the rows for the order date, which is 2003 Quarter 1. This way, full table scans are limited to a specific partition only, saving significant I/O. When the optimizer chooses to scan only some partitions and not all, this is known as "partition pruning" or "elimination."

But that is a basic property of partitioning — nothing new there. The important question is, how you can ensure that the partition pruning or elimination has indeed occurred? You can do so by explaining the query first and querying the PLAN_TABLE. Consider a table created as follows:

```
create table ptest1
(
col1 number,
col2 varchar2(200),
col3 varchar2(200)
)
partition by range (col1)
(
partition p1 values less than (1001),
partition p2 values less than (2001),
… and so on
partition p9 values less than (9001),
partition pm values less than (maxvalue)
);
```

Now, we will insert several records into this table so that each partition will have at least one record; then, we'll analyze the table:

```
insert into ptest1
select rownum, object_type, object_name
from all_objects
where rownum < 10001;
commit;
```

Next, we'll examine the optimization plan for a query that will be issued on the table PTEST1 as follows:

```
EXPLAIN PLAN FOR
SELECT * FROM PTEST1
WHERE COL1 = 1500;
```

This populates the PLAN_TABLE with the optimization plan records. Now, we'll choose the plan using the "SELECT ..." query. (Note: To perform the actions shown in this article, you will be using this query a lot; you should save the query in a script named plan.sql. The column FILTER_PREDICATES will be found only in Oracle9i, so remove the column from this query when running against an Oracle8i database.

```
select id, lpad(' ',level*1-1)||operation||' '||options||' on
'||object_name operation,
partition_start PB, partition_stop PE,
partition_id, filter_predicates
from plan_table
connect by parent_id = prior id
start with parent_id is null;
```

The result is as follows:

```
ID OPERATION                       PB PE  PI
--- ---------------------------- -- -- ---
FILTER_PREDICATES
--------------------
  0 SELECT STATEMENT  on

  1  TABLE ACCESS FULL on PTEST1    2  2    1
"PTEST1"."COL1"=1500
```

This could have been done via DBMS_XPLAN.DISPLAY(), too, but to make it version independent, we'll use PLAN_TABLE. Look at the PARTITION_START and PARTITION_STOP columns; values are both 2, indicating that the data will be selected from partition number 2 only. This is not expected, since the value 1500 will be available in partition 2 only. How does the optimizer know which partition to look for? It does so at the Step ID 1 in the optimization plan as indicated by the column PARTITION_ID in *plan_table*.

Finally, we also know that the optimizer applied a filter to retrieve rows as in the column FILTER_PREDICATES. This explains how the optimizer came up with the plan and from which segments it will select. This type of analysis will be most helpful when you are testing the different partition pruning scenarios.

Let's introduce another complexity to the mix — subpartitioning. Consider a table created as follows:

```
create table ptest2
(
col1 number,
col2 varchar2(200),
col3 varchar2(200)
)
partition by range (col1)
subpartition by hash (col2)
subpartitions 4
(
partition p1 values less than (1001),
partition p2 values less than (2001),
and so on…
partition p9 values less than (9001),
partition pm values less than (maxvalue)
);
```

We will insert rows in the same manner as the example used previously, and analyze the table. Then we will issue the query as follows:

```
EXPLAIN PLAN FOR
SELECT COL2 FROM PTEST2 WHERE COL1 = 9500
AND COL2 = 'PROCEDURE';
```

Here, the query is forced to select from a subpartition, as the filter is based on the partitioning as well as the subpartitioning key. The query on PLAN_TABLE shown earlier displays the following output:

```
  ID OPERATION                     PB PE  PI
---- ----------------------------- -- -- ---
FILTER_PREDICATES
--------------------------------------------
   0 SELECT STATEMENT  on

   1  TABLE ACCESS FULL on PTEST2  38 38   1
"PTEST2"."COL1"=9500 AND "PTEST2"."COL2"='PROCEDURE'
```

Note the PARTITION_START column; it shows 38 — but we don't have that many partitions. Actually, the number 38 reflects the count of subpartitions, not partitions. In this example, the number of subpartitions in a partition is four, so the first nine partitions in the table contain the first 36 subpartitions. The thirty-seventh and the thirty-eighth subpartitions exist in the tenth partition. The tenth partition is the partition PM, making the highlight subpartition the second one under that. If you look at the query, the optimizer correctly selected the partition PM for elimination.

Make note of this feature to avoid confusion - the PARTITION_START and PARTITION_STOP columns also point to subpartitions, if they are involved, not just partitions.

Partition-wise Joins

When a partitioned table is joined to another partitioned table in such a way that partitioning keys determine the filtering, the optimizer can determine that it does not need to search the whole table, but just the partitions in which the data resides. For instance, consider the table SALES range, partitioned on the SALES_DATE column, the table REVENUE range, partitioned on the BOOKED_DATE column, and the partitioning schemes (the boundary values of the partitions are the same). These tables are "equi-partitioned." If the user queries using the following:

```
SELECT … FROM SALES S, REVENUE R
WHERE S.SALES_DATE = R.BOOKED_DATE
AND S.SALES_DATE = '31-JAN-2003';
```

then the optimizer knows that the rows returned by the filtering condition, SALES_DATE = '31-JAN-2003' will be found only in a single partition, the one for 2003 Quarter 1. Since the REVENUE table is equi-partitioned, the rows also will be found only in that table's partition for 2003 Quarter 1.

So, for each row in SALES, only rows in a particular partition in REVENUE need to be searched, not the entire table.

Next, we will examine if such a process of selection is indeed happening. Consider two tables created as follows:

```
create table ptest3a
(
col1a number,
col2a varchar2(200),
col3a varchar2(200)
)
partition by range (col1a)
(
partition p1 values less than (1001),
partition p2 values less than (2001),
and so on…
partition p9 values less than (9001),
partition pm values less than (maxvalue)
);

create table ptest3b
(
col1b number,
col2b varchar2(200),
col3b varchar2(200)
)
partition by range (col1b)
(
partition p1 values less than (1001),
partition p2 values less than (2001),
and so on…
partition p9 values less than (9001),
partition pm values less than (maxvalue)
);
```

Note the tables have been range partitioned in an identical manner. Next, we will insert data into both tables so that all partitions will have at least one row, as follows:

```
insert into ptest3a
select rownum, object_type, object_name
from all_objects
where rownum < 10001;

insert into ptest3b
select rownum, object_type, object_name
from all_objects
where rownum < 10001;
```

After analyzing both tables, a user queries the tables in this manner:

```
explain plan for
select count(*)
from ptest3a , ptest3b
where ptest3b.col1b = ptest3a.col1a
and ptest3a.col1a between 1500 and 1700;
```

and then queries from the PLAN_TABLE using the script plan.sql, she gets

```
  ID OPERATION                      PB PE  PI
---- ----------------------------- -- -- ---
FILTER_PREDICATES
--------------------
   0 SELECT STATEMENT  on
   1  SORT AGGREGATE on
   2   NESTED LOOPS  on
   3    TABLE ACCESS FULL on PTEST3A  2 2   3
 "PTEST3A"."COL1A">=1500 AND "PTEST3A"."COL1A"<=1700
   4    TABLE ACCESS FULL on PTEST3B 2  2   4
"PTEST3B"."COL1B"="PTEST3A"."COL1A" AND  "PTEST3B"."COL1B">=1500 AND
"PTEST3B"."COL1B"<=1700
```

Note how only partitions with ID# 2 from each table were subjected to Full Table Scans, not the entire table; this enabled partition-wise joins. The optimizer determined that partition-wise joins are possible in step ID 3 and step 4, as shown in the column PARTITION_ID. And it knew which partitions to join from the filter predicates, easily explained in the output. Since the rows will be found in partition ID 2 only, only that partition of ptest3a is used. And, since ptest3a and ptest3b are equi-partitioned, the optimizer will search for rows only in partition ID 2 of ptest3b, too, not the entire table.

Now let's see how a different type of partitioning scheme, hash partitioning, behaves for partition-wise joins. Consider the following two tables:

```
create table ptest3a
(
col1a number,
col2a varchar2(200),
col3a varchar2(200)
)
partition by hash (col1a)
partitions 4;

create table ptest3b
(
col1b number,
col2b varchar2(200),
col3b varchar2(200)
)
partition by hash (col1b)
partitions 4;
```

So each table has 4 hash partitions. Insert the data in the same way as in the previous example and analyze. If we explain the same query as we did before, and select from the plan table, we get

```
  ID OPERATION                        PB PE PI
---- ------------------------------ -- -- --
FILTER_PREDICATES
--------------------
   0 SELECT STATEMENT  on
   1  SORT AGGREGATE on
   2   PARTITION HASH ALL on           1  4  2
   3    NESTED LOOPS  on
   4     TABLE ACCESS FULL on PTEST3A 1  4  2
"PTEST3A"."COL1A">=1500 AND "PTEST3A"."COL1A"<=1700
   5     TABLE ACCESS FULL on PTEST3B 1  4  2
"PTEST3B"."COL1B"="PTEST3A"."COL1A" AND "PTEST3B"."COL1B" >=1500 AND
"PTEST3B"."COL1B"<=1700
```

Note the partition start (1) and stop (4) values, which are for *all the* partitions. This query does *not* perform a partition-wise join; it simply scans the entire table, even though it could have eliminated certain partitions. The filter predicates indicate that the optimizer knew about the rows to look for. So why didn't it do a partition-wise join?

The problem is the way hash-partitioned tables handle joins. In this example, the filtering condition is a range, between 1500 and 1700, not a specific value. This means the optimizer will not be able to point to a single partition for selection of the rows, and therefore a full-table scan is necessary. Partition-wise joins will not occur in this case. Let's take a look at another variation of this query:

```
explain plan for
select count(*)
from ptest3a , ptest3b
where ptest3b.col1b = ptest3a.col1a
and ptest3a.col1a = 1500;
```

Note the filtering predicate has been changed from a "between" to an "equality" with a constant. Using the *plan.sql* script, we get the "explain plan" as

```
ID OPERATION                                  PB PE  PI
---- ---------------------------------------- -- -- ---
FILTER_PREDICATES
--------------------
0 SELECT STATEMENT on
1  SORT AGGREGATE on
2   NESTED LOOPS on
3    TABLE ACCESS FULL on PTEST3A              3 3   3
"PTEST3HA"."COL1A"=1 500
4    TABLE ACCESS FULL on PTEST3B              3 3   4
"PTEST3HB"."COL1B"="PTEST3A"."COL1A" AN D "PTEST3B"."COL1B" =1500
```

The PARTITION_IDs for start and stop partitions are 3 each, as expected. This means the third partitions of both table have been joined to get the answer; in other words, we have just achieved a partition-wise join. How were we able to do this?

If the filter predicates are based on equality operator only, then the optimizer can assign a specific partition to the predicate by using the hash function. That is why the partition-wise join was possible in the second example, but not in the first example. If the predicate is a range, the optimizer cannot decide whether a particular partition may be a candidate. Be very careful in designing hash-partitioned tables when there is a chance of joining with range filtering.

Character Value in Range Partitioning

Almost all documents, articles, books, and other documentation talks about range partitioning using either dates (the most common) or numbers. However, the partitioning scheme could be extended to character strings too. Consider the example of the employee table where the last name column is the partitioning key, to separate employees into multiple partitions by last name. Consider a table where the first partition P1 should hold all last names starting with C and below, P2 should hold between D and F; finally the rest with partition PM. According to a MetaLink Note, here is the proper syntax for designing such a partitioning scheme.

```
CREATE TABLE EMP (…………)
PARTITION BY RANGE (LAST_NAME)
(
PARTITION P1 VALUES LESS THAN ('D%'),
PARTITION P2 VALUES LESS THAN ('G%'),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

Note the percentage character after the names. This ensures that the ranges are well delineated by the boundaries. Consider this example:

```
SELECT * FROM EMP;

LAST_NAME  FIRST_NAME
---------- ----------
CHAPLIN    CHARLIE
D          HARLEY
DAVIDSON   HARLEY
EINSTEIN   ALBERT

SELECT * FROM EMP PARTITION (P1);

LAST_NAME  FIRST_NAME
---------- ----------
CHAPLIN    CHARLIE
D          HARLEY
```

```
SELECT * FROM EMP PARTITION (P2);

LAST_NAME  FIRST_NAME
---------- ----------
DAVIDSON   HARLEY
EINSTEIN   ALBERT
```

Note the placement of two rows with last names starting with D. The last name DAVIDSON is placed in P2 as expected, but the last name D is placed in partition P1. Shouldn't DAVIDSON and D be within the same partition, P2?

Actually, this is not unusual. The character set comparison, "D" is less than "D%," satisfying the boundary of the partition P1, and that is where the last name "D" goes, even though you probably expected the last name "D" to go into the same partition as DAVIDSON. While designing such a partitioning scheme be mindful of the potential problem.

Consider the same table in a slightly different way:

```
CREATE TABLE EMP (…………)
PARTITION BY RANGE (LAST_NAME)
(
PARTITION P1 VALUES LESS THAN (‘D’),
PARTITION P2 VALUES LESS THAN (‘G’),
PARTITION PM VALUES LESS THAN (MAXVALUE)
);
```

Note, there is no percentage sign after the character values. Inserting the same data into it and selecting from different partitions, we get

```
SELECT * FROM EMP2 PARTITION (P2);

LAST_NAME  FIRST_NAME
---------- ----------
DAVIDSON   HARLEY
EINSTEIN   ALBERT
D          HARLEY
```

Note how the partition P2 now has both DAVIDSON and D. This modified approach will help you avoid potential problems in the future. If you design character-based range partitioning, you should consider dropping the percentage character in your boundary to eliminate confusion, even though it is as specified in the MetaLink Note. If you use Oracle9i, you can probably change most of your character-based partitioning schemes to LIST.

This article should provide some insights into some potentially problematic situations regarding partitioning. More potential problems and pitfalls will be discussed in the Part 2 of this article next month.

Effective Segment Partitioning – Part 2

CHAPTER 24

Perils and Pitfalls in Partitioning — Part 2

This is a continuation of last month's article on partitioning. In case you haven't seen the first part, here it is. Partitioning is a favorite topic for authors, presenters, and for the DBA community in general, but most of the papers on this subject dwell on the basics and fundamental concepts behind partitioning. The inevitable action of most DBAs, after learning the ropes, is to jump into their databases with partitioning in mind. But not so fast. This article describes some the potential problems with partitioning, features with little, or no, documentation that may create unforeseen situations, and how to resolve these. To get the most out of this article, you should already have some basic knowledge about partitioning — this is not a primer.

Multi-Column Partition Keys

Most documentation, articles, books, and so on, talk about a single column as a partitioning key, but how about two or more columns in the partitioning key? It's definitely possible, but in such a case, how should you proceed?

Many people are under the impression that specifying more than one column as partitioning key creates a multi-dimensional partitioned table. For example, if you have a table called "employee range" partitioned on (DEPTNO, ZIPCODE), does that mean that the values of both columns are evaluated when you are deciding about the placement of the row in a partition?

Unfortunately, the answer is - no.

The second column in the partitioning key is used only in some special cases. Both values do not need to be satisfied for an insert to go to a specific partition. The first column is evaluated first; if it satisfies the condition, then the second column is *not* evaluated. However, if the first column value is borderline satisfactory, the next column is considered.

This is perhaps better explained using an example. Consider the following:

```
create table ptab1

    col1 number(10),
    col2 number(10),
    col3 varchar2(20)
)
partition by range (col1, col2)
(
partition p1 values less than (101, 101),
partition p2 values less than (201, 201)
)
```

It is a popular perception that when a row is inserted, if the values of col1 and col2 both are less than 101, then it goes to partition P1; if the values are less than 201, but more than or equal to 101, it goes to partition P2; otherwise, it goes to partition PM. In our example, let's see which partition holds what. Here are all the rows of the table:

```
select * from ptab1;
COL1             COL2 COL3
---------- ---------- ------
       100        100 rec1
       102        102 rec2
       100        102 rec3
       102        100 rec4
       101        100 rec5
       101        101 rec6
       101        102 rec7
       201        100 rec8
       201        101 rec9
       201        102 rec10
```

In which partitions do you think the records will be? Let's check the first one:

```
select * from ptab1 partition (p1);
COL1             COL2 COL3
---------- ---------- ------
       100        100 rec1
       100        102 rec3
       101        100 rec5
```

Record REC1 is in partition P1 as expected. But should REC3 be in partition P1? The value of column COL1, which is 100, is less than 101 and therefore satisfied. But COL2 is 102, and is more than 101, the boundary value of COL2. How does COL2 end up in the P1 partition? The reason is quite simple: P1 is the first partition, it's evaluated for the first column (COL1), the value satisfies it, so the value of column COL2 is *not even evaluated*. The record goes to P1, *even though* COL2 is not satisfied.

So, if the second column, COL2, is not even considered at all in some cases, where does it come into play and why would you define it? Consider the record REC5, in which the COL1 value is 101, a borderline value of that column in the partitioning key. But in this case, the second column is considered. In this case, COL2 value is 100, less than the boundary value of COL2 in the partitioning key (101); therefore, it goes into the partition P1. Look at the records in partition P2.

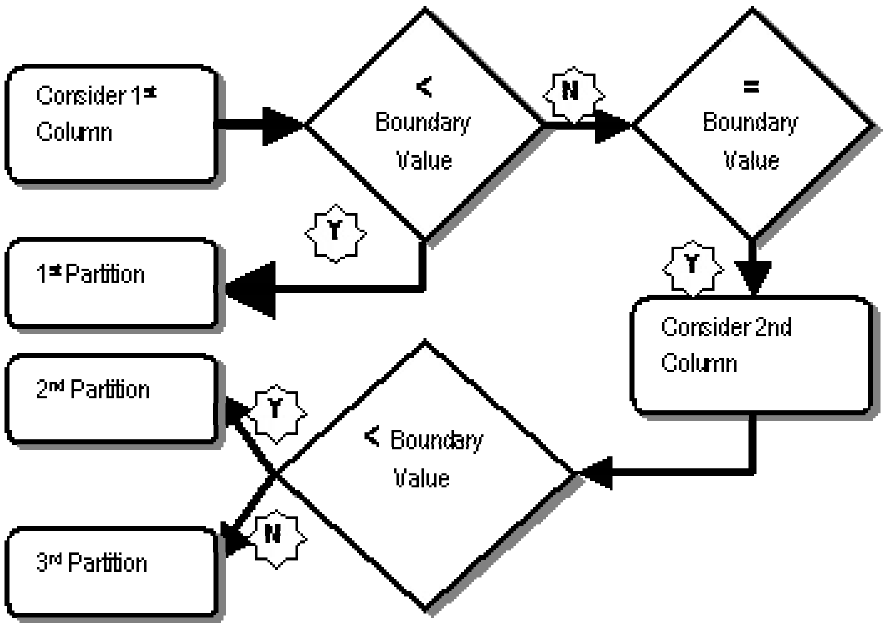
```
select * from ptab1 partition (p2);
COL1             COL2 COL3
---------- ---------- -----
       102        102 rec2
       102        100 rec4
       101        101 rec6
       101        102 rec7
       201        100 rec8
       201        101 rec9
       201        102 rec10
```

The records REC2, REC4, and REC7 satisfy both columns and are as expected in partition P2. However, for REC6, the COL1 value is 101, which is the boundary value for first column of the partitioning key. So, REC6 falls under the special consideration for multi-column partitioning keys. Because the COL2 column value of 101 is more than the boundary value of column COL2 of partition P1 (101), the rows went to partition P2.

In the same logic, for records REC8, REC9, and REC10, the COL1 value is 201 — right on the boundary for the value of that column in the partitioning key. However, the value of COL2 is less than 201, and the boundary value of that column in P2. Therefore, the rows went to partition P2.

What happens when you insert a row with COL1 = 201 and COL2 = 201?

That row will go into partition PM, since both columns cannot be outside the bounds. Schematically, the decision to insert into a partition can be explained as in the figure below.



So what happens in the case of list partitioning in Oracle 9i, when there is no concept of a range, so there is no boundary value? Fortunately, list partitioning does not allow multiple columns, so this situation does not arise.

It seems that, given the potential confusion about the placement of rows in partitions, it's not worth pursuing the use of multi-column partitioning keys. However, in some special cases, it can be very useful. Consider a table called SALES, for instance, with columns SALES_YEAR, SALES_MONTH and SALES_DAY, instead of a single column called SALES_DATE. This is useful in some data warehouse design implementations to enable dimensions and hierarchies. In such a case, you could use a partitioning key in all three columns to effectively design the partitions.

Potential Pitfall: Be careful while defining multiple columns as partitioning keys. If you must do so, use test cases exactly around the boundary values.

Subpartition Statistics

This is one tricky part of subpartitioning, which is not well documented and clear in the manuals. You must have been using the DBMS_STATS package for quite some time now to collect statistics. To collect statistics for the tables and the sub-objects under them (e.g. , partitions and subpartitions), you should use the function under the package named GATHER_TABLE_STATS. The function has two, little-known parameters that must be set for proper statistics collection.

PARTNAME

This parameter is supposedly set to collect statistics for only the named partition within the table, not for the entire table. However, this is a misconception. PARTNAME can be used to collect the stats for a specific subpartition, too. In order to do that, the name of the subpartition is passed as this parameter.

GRANULARITY

This parameter instructs the package to collect statistics at different levels and to cascade down to other sub-objects. It accepts several values. The default, named DEFAULT, instructs the package to collect global statistics and on the partitions only. The PARTITION value instructs the package to collect stats at the partition level. However, setting these values will not collect stats at the *subpartition* level; these can be collected by setting the parameter to ALL or SUBPARTITON.

Consider the table created as follows:

```
create table spart1
(
   col1    number,
   col2    number,
   col3    varchar2(20)
)
partition by range (col1)
subpartition by hash (col2)
subpartitions 4
(
   partition p1 values less than (101),
   partition p2 values less than (201),
   partition p3 values less than (301),
   partition p4 values less than (401),
   partition pm values less than (maxvalue)
)
```

Analyze the table using the default value of granularity as follows:

```
exec dbms_stats.gather_table_stats (tabname=>'SPART1')
```

Note, we have not provided the granularity at all. Since the default value is to collect stats for the partitions only, and not for any of the subpartitions, the stats will not be collected for the subpartitions. This can be verified by issuing:

```
select partition_name
from user_tab_subpartitions
where last_analyzed is not null;
```

This command will not return any rows. But let's analyze the other options here. A table can have statistics at the table level only, called GLOBAL statistics. If the partitions of the table are analyzed and the optimizer can derive the global statistics from the individual partitions, then the stats for the table are supposed to be derived globally. Let's examine each option in detail:

```
exec dbms_stats.gather_table_stats (tabname=>'SPART1',
granularity=>'GLOBAL')
```

This collects stats at the global level only. The following query confirms this.

```
Select last_analyzed, global_stats
From user_tables where table_name = 'SPART1';
```

This returns

```
GLO LAST_ANAL
--- ---------
YES 10-MAR-03
```

The presence of global stats indicates that the table has been analyzed as a whole, but the optimizer will not know the stats of individual partitions. This can be gathered using:

```
exec dbms_stats.gather_table_stats (tabname=>'SPART1',
granularity=>'PARTITION')
```

This command sets the stats at the partition level only. In this case, the global stats are not collected on the table, and the query above will return a NO under GLOBAL_STATS. However, the query

```
select partition_name, last_analyzed
from user_tab_partitions
where last_analyzed is not null;
```

will retrieve all the partitions. Another variation of the package is shown below.

```
exec dbms_stats.gather_table_stats (tabname=>'SPART1',
granularity=>'SUBPARTITION')
```

This collects stats on the subpartition level only, and infers the stats on the partition level; however, it does not collect global stats on the partitions itself.

The last value of the option, ALL, performs all of these — collects partition-level, and subpartition level stats, as well as the global stats on the subpartition, partition, and table.

Thus, the default value for the *granularity* parameter in the stats gathering function does not collect stats on subpartitions; you must set it to either SUBPARTITION or ALL to gather stats.

In summary, here are the details about setting granularity and collecting statistics:

GRANULARITY	TABLE GLOBAL	PARTITION GLOBAL	PARTITION STATISTICS	SUB-PARTITION STATISTICS
GLOBAL	YES	NO	NO	NO
PARTITION	NO	YES	YES	NO
DEFAULT	YES	YES	YES	NO
SUBPARTITION	NO	NO	YES	YES
ALL	YES	YES	YES	YES

Another interesting concept that is not documented clearly is the option to analyze subpartitions only. This can be done using:

```
exec dbms_stats.gather_table_stats (tabname=>'SPART1',
PART_NAME=>'P1_SYS123')
```

This will collect subpartition-level stats on subpartition P1_SYS123 only.

Rule Based Optimizer

Can you use partitioning with Rule Based Optimizer (RBO)? The answer is, of course you can. However, when partitioning was introduced, RBO was considered legacy, and Oracle decided to gradually phase out support for it. This led to a general stop in development of RBO, so today, RBO is not set up to exploit several exciting developments, partitioning included. Therefore, to get the full advantage of partitioning (partition pruning, partition-wise joins, and so on), you must use the Cost Based Optimizer (CBO). If you use the RBO, and a table in the query is partitioned, Oracle kicks in the CBO while optimizing it. But because the statistics are not present, the CBO *makes up* the statistics, and this could lead to severely expensive optimization plans and extremely poor performance.

So, although you can, you shouldn't use partitioning when using the RBO.

Coalesce vs. Merge

These two potentially confusing statements serve the same purpose — reducing the number of partitions – and are applicable in different schemes. In a range- or list-partitioned table, the partition boundaries are clearly defined, and the rows in a partition satisfy some condition dependent on the boundary values. ALTER TABLE … MERGE PARTITION joins the two adjacent partitions and sets the boundary values appropriately.

Consider the example of a table PART that is partitioned by range into four different partitions named P1, P2, P3, and P4.

To merge partitions P3 and P4 to make a partition called P34, issue the following statement:

```
ALTER TABLE PART MERGE PARTITIONS P3, P4 INTO PARTITION P34;
```

However, in hash-partitioned tables, there are no boundary values, and the rows are not decided as candidates for the partitions based on some kind of defined range. So, a merge will not be able to identify and set specific boundaries. You should use a new clause called COALESCE to achieve this objective:

```
ALTER TABLE PART COALESCE;
```

In COALESCE, a specific partition, usually the last one, is identified for elimination. All the rows in that partition are supposed to be equally distributed over the remaining partitions and the partition is dropped. In practice, however, the rows are merged with the adjacent partition.

Since this reduces the number of partitions by one, the total number is not a power of two any more, making the distribution of rows in all partitions unequal. To avoid this problem, issue the COALESCE one more time to make the partitions evenly loaded.

In summary, MERGE is for range and list partitioning when the values are clearly identified for boundary values, and COALESCE is for hash partitions, to reduce the number of partitions.

Other Questions

What about Rebuild Partition and Global Indexes?

Oracle9iR2 now offers fast split partitioning. Typically, during a split operation, Oracle creates two new partitions and then redistributes the rows from the source partition to the new partitions. This is a very expensive operation from the resource consumption point of view. In addition, local index partitions become unusable.

With fast split partitioning, if all the rows will exist in the same partition after the partition split, Oracle simply reuses the old partition and creates an empty partition. Thus, a split action becomes more like a complete operation that just creating a new partition.

Global indexes become unusable when a partition is rebuilt. However, in 9i, a new clause updates the global indexes as well.

```
ALTER TABLE PTAB DROP PARTITION P2 UPDATE GLOBAL INDEXES;
```

While using partitioning, should you use bind variables?

This is an interesting question. As we all know, use of bind variables eliminates the need to parse the cursors and makes it easier to reuse the cursors.

In case of partitions, however, using bind variables poses a problematic situation. Partition elimination and joins can occur only if the optimizer knows the filtering predicate in advance. The value of bind variables are not known until it's time to execute, making the process of partition elimination or joins impossible. Therefore, to take advantage of these options, you should not use bind variables.

In Oracle 9i, the first parse of the statement, called hard parse, peeks into the value of the bind variable, and can effect these optimization options. But this occurs only with the hard parse; subsequent parses still go around the bind variable values.

How many partitions can be defined on a table?

Oracle uses a two-byte field to store the number of segments (partitions or subpartitions), which enables 2^16 or 65536 spaces. The Oracle code, therefore, allows one fewer than this number — 65535. Note that this is a limit set by Oracle software code; an actual limit may be lower.

Remember, every time a query is parsed on a partitioned object, the metadata (i.e., how many partitions, and so on) is loaded into the cursor cache in SGA, meaning the SGA should be large enough to handle a table with several partitions.

Multi-Master Replication

CHAPTER 25

A Four-phase Approach to Procedural Multi-master Replication

Introduction

Do you support customers whose databases are updated by users in multiple locations and across multiple time zones? If so, the challenge for the DBA is how best to manage replicated systems that allow for fast database access over Wide Area Networks.

In many shops, popular failover solutions include Real Application Clusters (RAC) and Oracle9i Dataguard. An alternate solution, however, is growing in popularity: Oracle advanced replication — specifically, procedural multi-master replication.

With Oracle multi-master replication, you can implement peer-to-peer replication of all master tables, anywhere in the world. You can update any master site by propagating changes, either synchronously or asynchronously, and apply those changes directly to all other master tables. In addition to providing fast database access across your WAN, multi-master replication also provides solutions for failover and load-balancing issues.

What's the catch? Multi-master replication is extremely sophisticated and complex process. You can configure an almost infinite array of multi-master replication models, each adhering to its own set of conflict resolution and refresh rules. You may have heard that advanced replication implementations are notoriously difficult to configure, and they are. Large Oracle shops may spend hundreds of hours configuring and testing a worldwide multi-master replication solution, and many have a dedicated DBA whose sole job is to monitor and maintain the multi-master replication. In the long run, however, the investment in time and resources is worth the extra effort.

In this article, I'll present several reasons why multi-master replication is popular for geographically distributed systems. For those of you who are new to the basic concepts of multi-master replication, I'll present a high-level explanation of how it works, including code samples. Then we'll look at a four-phase plan for implementing procedural multi-master replication. Finally, I'll tell you where to find three pre-defined PL/SQL packages from Oracle that help define multi-master replication.

Why Consider Oracle Multi-master Replication?

There are a couple of reasons why Oracle multi-master replication is so popular for geographically distributed systems. Perhaps the most important reason is that it provides multiple-node replication capabilities. This may seem obvious, but you must remember that one-way read-only snapshots are far easier to create and maintain than a multi-master scheme.

The other benefit of multi-master replication is the ability to replicate stored procedures. In a system where all code is encapsulated inside Oracle stored procedures, you can replicate the stored procedures to remote sites, just like data. This capability allows the DBA to coordinate code changes with database changes. Once the Oracle stored procedures are written, you can easily replicate and distribute them to work groups and branch offices throughout the entire replicated network of systems.

Oracle Multi-master Replication

I'll start with a high-level view of multi-master replication and introduce some basic concepts. Multi-master replication is such a complex topic that I can't fully address every issue about it in this space. However, I hope you'll be happy with a conceptual explanation of the mechanisms.

In a nutshell, multi-master replication is nothing more than a coordinated set of updateable snapshots. By "updateable," I mean that the snapshot allows the FOR UPDATE clause in the snapshot definition. To illustrate this concept, refer to the example below, where you'll see that the snapshot is allowed to propagate updates back to the master table.

```
create snapshot
   customer_updatable_snap
refresh fast start with sysdate
next sysdate + 1/24
for update
query rewrite
  as
   select * from customer@master_site;
```

Multi-master Conflicts and Resolutions

At first blush, multi-master replication may appear straightforward. However, there is a dark side to the process. Whenever a snapshot has the ability to send updates to other "master" tables, you always run the risk of update conflicts. So what's the best way to avoid and/or resolve those conflicts? Let's start the lesson by reviewing multi-master conflict avoidance. Then we'll dive head-first into the details of procedural replication, so we can see how it all fits together.

An update conflict occurs when one remote user overlays the updates made by a user on another database. Your multi-master replication model should detect and resolve conflicts. Unfortunately, detecting and resolving those conflicts can get extremely complex. Let's start by looking at what conflicts can occur, and then we'll look at mechanisms for resolving them.

Conflict Types

Here are the most common types of conflicts you'll encounter with multi-master replication:

- Uniqueness conflict — This conflict results from an attempt from two different sites to insert records with the same primary key. To avoid uniqueness conflicts, you can choose from three available options. Those three pre-built methods are called Append Site Name To Duplicate Value, Append Sequence To Duplicate Value, and Discard Duplicate Value.
- Update conflict — This conflict is caused by simultaneous update operations on the same record.
- Delete conflict — This type of conflict occurs when one transaction deletes a row that another transaction updates (before the delete is propagated).

Oracle provides several pre-written scripts to help in resolving conflicts. In the case of update conflicts, your only option is to write conflict-resolution routines, and deal with each conflict on a case-by-case basis. Fortunately, Oracle provides several pre-built methods for creating the routines

Conflict Resolution Mechanisms

Here are the most common mechanisms at your disposal for resolving conflicts:

- Latest Timestamp Value. With this simple technique, you apply updates as they are received. Based on timestamp value, the most recent updates overlays prior updates. This approach can result in situations where one user's update gets overlaid by a more recent update.
- Earliest Timestamp Value. This mechanism is the opposite of the latest timestamp value, in that the first update overlays subsequent updates. As you'd expect, not many shops use this method, but it is an option.
- Minimum and Maximum Value. This mechanism may be used when the advanced replication facility detects a conflict with a column group. The advanced replication facility calls the minimum value conflict resolution method and then compares the new value from the originating site with the current value from the destination site for a designated column in the column group. You must designate that column when you select the minimum value conflict resolution method.
- Additive and Average Value. When you're dealing with replicated numeric values, this additive method adds a new value to the existing value using the following formula: (current value = current value + (new value - old value)). The average method averages the conflicting values into the existing value using the formula (current value = (current value + new value)/2).
- Groups priority Value. Using this method, some groups have priority (a higher rank) over other groups. Therefore,

the update associated with the highest-ranked group gets the update.

- Site Priority Value. In this method, all master sites are NOT created equal. Some remote sites will have priority over other sites.

To illustrate how conflict resolution is defined, consider the example below. In this code, we execute *dbms_repcat.add_update_resolution* to direct Oracle to use the "latest timestamp" method for conflict resolution for updates to the EMP table.

```
execute dbms_repcat.add_update_resolution( -
        sname => 'SCOTT',                   -
        oname => 'EMP',                     -
        column_group => 'EMP_COLGRP',       -
        sequence_no => 1,                   -
        method => 'LATEST TIMESTAMP',       -
        parameter_column_name => 'EMPNO');
```

At this point, you should be starting to appreciate the complexity of conflict resolution in multi-master replication. Now let's take a quick look at the techniques you can use to define procedural multi-master replication.

Implementing Procedural Multi-master Replication

Although Procedural multi-master replication is an extremely complex process, you can break down the basic steps for defining procedural replication into four phases:

- Phase I: Pre-configuration. (Set-up Oracle parameters and catalog scripts.)
- Phase II: Define the repadmin user and database links.
- Phase III: Create master database and refresh groups.
- Phase IV: Monitor the replication environment.

Let's take a close look at each phase in turn.

Phase I: Pre-configuration Steps for Multi-master Replication

Before you're ready to define a multi-master replication environment, there's a short checklist you need to deal with up front. For every site that will be participating in the replication, you must check the values of these parameters:

- Oracle parameters minimum settings
  - *shared_pool_*size=10m
  - *global_names*=true
  - *job_queue_processes*=4

  To check those values, run this script on your database:

```
select
   name,
   value
from
   v_$parameter
where
  name in (
   'job_queue_processes',
   'global_names',
   'shared_pool_size');
```

- You also must be sure that the following dictionary scripts have been run from ORACLE_HOME/rdbms/admin. The catalog.sql was run when you created your instance, and the catproc.sql script is for the procedural option in Oracle.
  - catalog.sql
  - catproc.sql

Phase II: Set-up REPADMIN User and Database Links

The following illustrates some of the main steps you'll follow in pre-creating the REPADMIN users and the required database links for multi-master replication. You should review these steps with great care.

```
REM Assign global name to the current DB
alter database rename global_name to PUBS.world;
REM Create public db link to the other master databases
create public database link NEWPUBS using 'newpubs';
REM Create replication administrator / propagator / receiver
create user
   repadmin
identified by
   repadmin
default tablespace
   USER_DATA
temporary tablespace
   TEMP
quota unlimited on
   USER_DATA;
REM Grant privileges to the propagator, to propagate changes to remote
sites
execute dbms_defer_sys.register_propagator(username=>'REPADMIN');
REM Grant privileges to the receiver to apply deferred transactions
grant execute any procedure to repadmin;
REM Authorize the administrator to administer replication groups
execute dbms_repcat_admin.grant_admin_any_repgroup('REPADMIN');
REM Authorize the administrator to lock and comment tables
grant lock any table to repadmin;
grant comment any table to repadmin;
connect repadmin/repadmin
REM Create private db links for repadmin
create database link newpubs
        connect to repadmin identified by repadmin;
REM Schedule job to push transactions to master sites
REM This will replicate every minute
execute dbms_defer_sys.schedule_push(        -
           destination => 'newpubs',         -
           interval => 'sysdate+1/24/60',    -
           next_date => sysdate+1/24/60,     -
           stop_on_error => FALSE, -
           delay_seconds => 0, -
           parallelism => 1);
REM Schedule job to delete successfully replicated transactions
execute dbms_defer_sys.schedule_purge( -
        next_date     => sysdate+1/24, -
        interval      => 'sysdate+1/24');
REM Test the database link
select global_name from global_name@newpubs;
```

Phase III: Create the Master Database and Refresh Groups

Once the repadmin user and the links are in place, you're ready to define the replication. Again, this is an extremely complex process. However, the following script will provide you with the general steps to get the work done.

```
connect repadmin/repadmin



REM Create replication group for MASTERDEF site
execute dbms_repcat.create_master_repgroup('MYREPGRP');

REM Register objects within the group
execute dbms_repcat.create_master_repobject('SCOTT', -
        'EMP', 'TABLE', gname=>'MYREPGRP');

execute dbms_repcat.make_column_group(   -
        sname => 'SCOTT',                -
        oname => 'EMP',                  -
        column_group => 'EMP_COLGRP',    -
        list_of_column_names => 'EMPNO');

execute dbms_repcat.add_update_resolution( -
        sname => 'SCOTT',                -
        oname => 'EMP',                  -
        column_group => 'EMP_COLGRP',    -
        sequence_no => 1,                -
        method => 'LATEST TIMESTAMP',    -
        parameter_column_name => 'EMPNO');

REM Add master destination sites
execute
        dbms_repcat.add_master_database( -
        'MYREPGRP', -
        'TD2.world');

REM Generate replication support for objects within the group
execute
        dbms_repcat.generate_replication_support( -
        'SCOTT', -
        'EMP', -
        'table');
```

Dropping Multi-master Replication

As you'd expect, there will be instances when you may need to turn-off multi-master replication. Some of the obvious cases include database maintenance activities such as upgrades and reorganizations. You can use this sample script to disable multi-master replication.

```
connect repadmin/repadmin


REM Stop replication
execute dbms_repcat.suspend_master_activity(gname=>'MYREPGRP');

REM Delete replication groups
-- execute dbms_repcat.drop_master_repobject('SCOTT',    'EMP',
'TABLE');
execute dbms_repcat.drop_master_repgroup('MYREPGRP');
execute         dbms_repcat.remove_master_databases('MYREPGRP',
'newpubs.world');
REM Remove private database links to other master databases
drop database link newpubs.world;


connect sys
REM Remove the REPADMIN user
execute
   dbms_defer_sys.unregister_propagator(username=>'REPADMIN');

execute

dbms_repcat_admin.revoke_admin_any_schema(username=>'REPADMIN')
;
drop user repadmin cascade;
REM Drop public database links to other master databases
drop public database link newpubs.world;
```

Phase IV: Monitoring Multi-master Replication

The final phase of implementing multi-master replication involves monitoring. A variety of dictionary views provide the key to monitoring complex multi-replication processes. I cannot stress enough the importance of checking these views on every database in the multi-master network.

- *dba_repschema*. This view contains details for the replication schema

- *dba_repcatlog*. This view provides a log of all replication activities.
- *dba_jobs*. Use this view to monitor all scheduled job in the database.
- *dba_repcat*. This view shows the replication catalog.
- *all_repconflict*. This view provides a list of all replication conflicts.
- *all_represolution*. For systems defined with pre-defined conflict resolution, this view lists the resolution of every conflict.
- *dba_repobject*. This view gives you a list of al replicated objects.
- *dba_repsites*. This view provides is a list of replicated sites.

At this point, you'll want to closely review the following script, which is the one most commonly used to monitor procedural replication. Of course, you must run this script on each remote database.

```
connect repadmin/repadmin
set pages 50000
col sname      format a20 head "SchemaName"
col masterdef  format a10 head "MasterDef?"
col oname      format a20 head "ObjectName"
col gname      format a20 head "GroupName"
col object     format a35 trunc
col dblink     format a35 head "DBLink"
col message    format a25
col broken     format a6 head "Broken?"
prompt Replication schemas/ sites
select
  sname,
  masterdef,
  dblink
from
  sys.dba_repschema;
prompt RepCat Log (after a while you should see no entries):
select
   request,
   status,
   message,
   errnum
from
   sys.dba_repcatlog;
prompt Entries in the job queue
select
   job,
   last_date,
   last_sec,
   next_date,
   next_sec,
   broken,
   failures,
   what
from
   sys.dba_jobs
where
   schema_user = 'REPADMIN';
prompt Replication Status:
select
   sname,
   master,
   status
from
   sys.dba_repcat;
prompt Returns all conflict resolution methods
select * from all_repconflict;
prompt Returns all resolution methods in use
select * from all_represolution;
prompt Objects registered for replication
select
   gname,
   type||' '||sname||'.'||oname object,
   status
from
   sys.dba_repobject;
select * from dba_repsites;
```

Resources for Defining Multi-master Replication

When it comes to defining multi-master replication for your shop, you don't have to start from scratch. Oracle offers the following pre-defined PL/SQL packages that can assist you:

- *dbms_repcat* package — This complex package provides over 50 stored procedures. Follow this link for a listing of the procedures in *dbms_repcat* (http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96612/d_repcat.htm#93762).

- *dbms_reputil* package — This package contains several stored procedures. Here is a list of the procedures in dbms_reputil (http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96612/d_reputl.htm).
- *dbms_defer_sys package* — This collection contains 19 replication procedures. Here is a list of the procedures in *dbms_defer_sys* (http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96612/d_defsys.htm).

Conclusion

In this brief introduction it is impossible to provide a comprehensive overview of this powerful utility. Rather, the intent of this article was to provide a simple overview of the important concepts and illustrate how multi-master replication is used within a distributed Oracle environment.

References

Constraints on updatable snapshots - http://www.orafaq.com/papers/rep8cons.doc

Oracle advanced replication setup & design tips - http://oracle.ittoolbox.com/browse.asp?c=OraclePeerPublishing&r=/pub/DS071702.pdf

Oracle FAQ - Oracle advanced replication scripts - http://www.orafaq.com/faqscrpt.htm#ADVREP

Oracle Magazine – "The Best Strategy for Disaster Recovery: Multi-Master Asynchronous Replication"

Oracle replication solutions -
http://www.dbasupport.com/oracle/ora9i/ors.shtml

Oracle8i advanced replication -
http://www.dbasupport.com/oracle/ora9i/ors.shtml

Oracle9i documentation - conflict resolution techniques -
http://www.engin.umich.edu/caen/wls/software/oracle/serve
r.901/a87499/repconfl.htm

Oracle9i Replication API documentation -
http://www.csis.gvsu.edu/GeneralInfo/Oracle/server.920/a9
6568/toc.htm

OracleNotes.com – Oracle8i advanced replication -
http://www.oraclenotes.com/Articles/Advance
Replication.ppt

Using updatable snapshots -
http://www.dbasupport.com/oracle/ora9i/snapshots.shtml

Replication Management

CHAPTER 26

Automated Replication Management

When and why did the replication fail? How many rows where updated at 3:25 am? If we load 100,000 new rows, how long will it take to replicate them all? Can the system handle this, or do we have to split it into smaller batches? Here is a PL/SQL package, which brings an automated solution to help with a number of replication problems like these.

The solution has been used on Unix (HP-UX 10.7/11, Sun Solaris 7/8, AIX 4, Linux 2) and Windows servers (NT4, 2000), on Oracle 7.3.x, 8.0.x, 8.1.x, 9.x. It requires some knowledge of UNIX shell scripts, SQLPlus scripts, and PL/SQL. However, full scripts are provided and minimal knowledge should be enough to start.

Basic Replication

Although advanced replication has been available for quite a while and has started to be used significantly, basic replication of simple materialized views (snapshots) with fast refreshes is still very much used in a very large number of installations for one-way transfers of information (mostly for production to data warehouse feeds, or for production to business-intelligence and web-enabled databases transfers, for central site to branches updates, for one database to another database updates, etc.).

Basic replication is the process of creating/maintaining a read-only copy (replica) object (table) in a local (secondary, slave) database, based on a read-write master object (table) in a remote (primary, master) database. The site hosting the master database is also called a "master site", and the site hosting the replicated schemas is called a "snapshot site". The replica, which is called a "materialized view" or a "snapshot", is built with a query very similar to a view. If the query references just one master table, and is simple enough the result is a SIMPLE SNAPSHOT. If the query references more master tables and/or contains DISTINCT or AGGREGATE functions, GROUP BY or CONNECT BY clauses, or some restricted types of SUBQUERIES, or JOINS, or SET OPERATIONS, then the result is a COMPLEX SNAPSHOT.

Only simple materialized views support FAST refreshes, complex materialized views have to be used with slower COMPLETE refreshes. If the snapshot log is not created, the system can perform only COMPLETE refreshes; if the refresh attempts the FAST method, it will fail. Only one snapshot log is possible per master table, even if several materialized views (from several slave schemas/databases) are referencing it.

See the comments in the *install-replisvs.sql* (http://www.dbazine.com/code/install-replisys.sql.txt) script (under P_RUN_SESSION) for a summary of the data dictionary changes in basic replication. (You can also see my article, "Diagnosing Oracle Replication Timings" in *Oracle Internals*, November 2002, for more details.)

Note: These are not documented facts, but conclusions inferred from studying a large amount of collected data in a replication environment! Most of them are based on the data dictionary base tables and are liable to change over Oracle versions. However, the package will continue to run correctly if the DBMS_REFRESH.REFRESH('group_name') command will remain unchanged; only some of the values in the associated tables might look strange.

Automated Replication Management

Our strategy was a combination of scheduled jobs (via crontab, etc. and/or DBMS_JOB) and a PL/SQL package (PKG_REPLISYS). Running the package takes up to 100 MB of memory and up to 20% CPU. The resource consumption is basically the same if the replication is run via the package or not. Space consumption is negligible.

Prerequisites

- you should have some system privileges (see the beginning of the *install-replisys.sql* (http://www.dbazine.com/code/install-replisys.sql.txt) script
- set *utl_file_dir* = * (or at least c:\temp, or /tmp, etc.) in *init.ora*, in order to allow log files to be created
- set *job_queue_processes* = 5 (or higher) in *init.ora*, in order to allow DBMS_JOB scheduling to work
- set *global_names* = FALSE in *init.ora*, in order to successfully run the creation of the test environment
- Unix user "oracle" should be allowed to write to '/tmp'
- *tnsnames.ora* entries to allow communications between the two databases (via database links)
- the package is installed on the snapshot site, and an MHSYS user has to exist in both databases

Associated Tables

Two identical tables (*replisys_list* and *replisys_hist*) hold identifying, processing and historical information. Some columns have values from the data dictionary, other columns contain calculated values.

Overview of the Package

The main problems that this solution addresses are:

- Oracle built-in replication packages do not record a failures history
- these packages do not record a performance history to be used for growth monitoring and capacity planning
- these packages do not raise exceptions in case of failure and only the server engine generates error messages

The Automated Snapshot Refresh package (PKG_REPLISYS) runs the DBMS_REFRESH.REFRESH('group_name') command and will also:

- record and calculate numbers and types of changes that need to be processed
- changes will include total/deletes/inserts/updates
- check that these changes have actually been processed
- record and calculate times for processing
- times will include session durations and statistically estimated snapshot refresh duration
- establish a performance baseline
- generate alerts if it detects high variations from the baseline
- generate alerts if it detects other errors

- detect and record some generic unavailability conditions
- try to overcome the limitation of the DBMS_REFRESH package of not raising exceptions

Initially, we build a few tables (see section ASSOCIATED TABLES), then we populate them with data from the data dictionary and calculated from running the package, with information about the processable objects (materialized views), which are sorted by group number. Based on a series of rules, the system then decides which object is assigned to which session. It starts with the first session, collects stats, runs the refresh command and collects stats again. The process continues until all objects in the session are processed. Then it records the session in the history table.

By repeating the process regularly the history table is populated with values that can be later used for various statistics, comparisons, performance troubleshooting, growth monitoring, capacity planning, etc. After a while you will know what are normal values for your system and use that as a baseline performance indicator.

When all sessions are done, we can examine the logs in the /tmp or c:\temp directories or in the history table. If there is no parameter in calling the package, the file based log will be generic and contain messages for all groups. If parameters are used in calling the package, numbered log files are created, with messages only for that particular group.

Upon completion, an email message can be sent to the DBA, or a grepping process can scan the logs and send an exit code/message to the operators, and the process is ready to start again.

When run manually in an SQLPlus session, display procedures ensure that debugging and detailed logging are made as easy as possible - currently many of these modules are commented out to avoid crashing the package because of overloading the server output buffer - uncomment them selectively.

Although it will not account for all situations, the package does log a wide variety of errors. The DBA will treat errors manually as the automated system will only try to re-run a session in case of failure. Some errors, like "ORA-12203: TNS: unable to connect to destination", which means the other database is not available, can be ignored, as it will probably clear on the next run (or you may have to page the DBA in charge of that database). Also, objects dropped after the list was created will cause benign errors. If the package is run automatically with 'DBMS_JOB', we get only a summary output (http://www.dbazine.com/code/DB2-REPLIsysPKG.log.txt), which can also include error messages.

Setup

The package is installed under the default Oracle user 'MHSYS', which I use to host my automation packages. It can be installed, as is, for Unix- and NT-based servers. It is a pretty comprehensive piece of software, which is compatible with Oracle 7.3.4 and later, on both UNIX and NT, and includes routines to detect the current OS, Oracle version and SID. It was successfully used on Oracle 7.3.4, 8.0.5, 8.0.6, 8.1.5, 8.1.6, 8.1.7, 9.0.1. It was also run successfully between databases of different versions.

It was run against tables of sizes up to 100,000,000 rows and 5GB. Sessions can vary between 1-60 minutes, depending on how often the refresh occurs and how large the refresh is. You should have the logs emailed to you or, at least, examine them manually. A grepping process can scan the logs for errors and issue exit codes for monitoring tools.

The code is amply commented. Run the *install-replisys.sql* (http://www.dbazine.com/code/install-replisys.sql.txt) script as user 'SYSTEM' from SQLPlus. Before installing, read the top of the package body, just in case you need to make some modifications. This section can also be used for tuning later, by changing the values of the constants. The defaults will cover most situations and, most likely, nothing will need to be changed. The install script can be run as is for Unix and NT based servers.

The code (2000 lines) performs a lot of error checking and decision-making in support of the refresh commands. Since you may want to run the refreshes in parallel for several groups, it accepts as parameter the group number or name and runs only one group per session, but more sessions simultaneously. If no parameter value is supplied, it will process all groups one by one, serialized. When supplied with a wrong group number or name, it exits with a message. Accepted commands are listed in the script *pkg_exec.sql.*(http://www.dbazine.com/code/pkg_exec.sql.txt)

You can use scripts to schedule or run the package, and to email the logs, similar to the ones described in my article "Automated Cost Based Optimizer" (*Oracle Magazine Online* - Sept 2000). For a list of Frequently Asked Questions and tips on running my packages, visit www.hordila.com/mhwork.htm.

Test Environment

The script *cre-test-replication.sql* (http://www.dbazine.com/code/cre-test-replication.sql.txt) will create a test replication environment for you to try this solution. To use the script, run it as is, interactively, or find and replace in the script "DB1" (master site) and "DB2" (snapshot site) with your test database names. This script will delete and recreate jobs with numbers: 301, 302, 303 in DB1, and 311, 312, 313 in DB2. The last section of the creation assumes that the package PKG_REPLISYS and associated objects have been already installed. This environment includes:

- in DB1 - one MHSYS schema (password MHSYS) for data dictionary queries for the package
- in DB2 - one MHSYS schema (password MHSYS) for the Replication Management Package PKG_REPLISYS
- in DB1 - 3 schemas (SNAPTEST1, SNAPTEST2, SNAPTEST3) - for the master tables
- in DB2 - 3 schemas (SNAPTEST1, SNAPTEST2, SNAPTEST3) - for the materialized view
- in DB1 - 6 master tables (TABLE1, TABLE2, …) - 3 with rowid, 3 with primary key - for each schema
- in DB1 - a total of 18 master tables per environment - 9 with rowid, 9 with primary key based replication
- in DB1 - 3 procedures to do inserts/deletes/updates automatically for each master table
- in DB2 - 6 materialized views (TABLE1R1, TABLE2R1, …) replicating from SNAPTEST1 to SNAPTEST1
- in DB2 - 6 materialized views (TABLE1R2, TABLE2R2, …) replicating from SNAPTEST2 to SNAPTEST2

- in DB2 - 6 materialized views (TABLE1R1, TABLE2R3, …) replicating from SNAPTEST3 to SNAPTEST3
- in DB2 - a total of 6 snapshots in 1 refresh group per user, and a total of 18 snapshots per environment
- in DB2 - a total of 3 groups with 6 materialized views each per environment
- we do not use the Oracle users (common replication administrators) REPADMIN and SNAPADMIN

The script *count_items_to_refresh.sql* (http://www.dbazine.com/code/count_items_to_refresh.sql.txt) from the snapshot site, finds the numbers of rows to be replicated. Run it after running the PKG_REPLISYS at least once, which creates automatically the correct database links.

Replication Master Table

CHAPTER 27

Altering the Master Table in a Snapshot Replication Environment without Recreating the Snapshot

Learn how to alter a master table in a read only or read write snapshot replication setup without dropping and recreating the snapshot or doing a full refresh, both of which can be extremely time and resource consuming. This leads to a time and effort savings of more than 98 percent.

One of the biggest challenges in administration of a snapshot replication environment (also called materialized view replication) is the usual maintenance of the snapshot after a modification of the master table. An example of this is adding columns or modifying the data type of a column. After a column is added to the master table, the only way the newly added column could be replicated to the replication site is by dropping the snapshot and recreating it. If the master table is large, the recreation process may take several hours as it brings all the data over the network. This also requires a large rollback segment both on the master and the replication sites and it may lead to the ORA-1555, "Snapshot Too Old" problem if the table access and rate of change is high. At a certain point, it may be impossible to even build the snapshot by recreating in this manner. At that point, the only option would be to do a full export and import of the table and recreate the snapshot by using the PREBUILT table option.

This problem, however, is not present in a multi-master setup. While the snapshot replication site presents numerous advantages in setting up and administration, this lack of ability to alter a master table easily poses real challenges to the DBAs maintaining the environment when they try to perform relatively trivial tasks like altering a table to add columns or change data types. This article presents a way to achieve these objectives and not having to recreate the snapshot or do a full refresh.

Background

To best illustrate the technique I'm presenting, let's start with an example: Suppose we have two databases, PROD and REPL, denoting production and the replication databases, respectively. All the activity happens on the production site whereas the replication site can be used as a reporting database only (read-only snapshot) or as a separate data activity point with the changes periodically pushed to master site (updatable snapshot). The technique described in this article applies to both situations. (Note: The snapshot replication is also called materialized view replication. In this example, the terms snapshot and materialized view are used interchangeably.)

Now suppose there are several tables under schema ANANDA. This example focuses on a table named TEST1, with about two million rows and four GB in total size. The table has two columns, COL1 NUMBER (9) and COL2 CHAR(1000), COL3 CHAR(900). The REPL database has a snapshot called TEST1 defined on the same table. The master site has a replication group called TEST1, which has only one object, TEST1. The replication group is owned by the schema REPADMIN. To set up the replication, use the script given in

Listing 1 (http://www.dbazine.com/code/Listing1.txt) at the master site. On the snapshot site, the snapshot TEST1 is included in a snapshot group called TEST1, owned by schema MVADMIN. It is assumed that there is a public database link from each database to the other in the same name and the global_names parameter in init.ora is set to true.

The goal is to alter the table TEST1 at master site adding a new column called COL4 with CHAR(1) and changing the column COL3 to CHAR(1000).

The Usual Method

For the purpose of demonstration, we first need to set up the replication environment. Listing 2 (http://www.dbazine.com/code/Listing2.txt) contains the statements to create the snapshot site. For convenience of discussion, the script is split into several sections, the most important of which, relevant to this article, is Section 3, Creating the Snapshot. This section actually creates the snapshot by getting the data across from the master site over the network. In case of a small table or during periods of light load, this approach might work without filling up the rollback segments or choking the network. However, in large tables, this method of transferring data might fail. Therefore, we have to follow a slightly different approach - using prebuilt tables.

In this approach, you have to drop the snapshot TEST1, if it exists, and create a table by the same name in the schema ANANDA with the same structure as that table at master site, but without data. This is easily done by running CREATE TABLE TEST AS SELECT * FROM TEST1@PROD WHERE 1 = 2. After running the script in Listing 1 (http://www.dbazine.com/code/Listing1.txt) at the master, you have to export the table from PROD and import into REPL. This might take quite awhile, but it will take considerably less time than the time required for the snapshot creation method. The table data can be brought over by other methods, too; e.g., by creating a delimited text file from the production database and loading it into the replication database using the SQL*Loader DIRECT path option.

Once the table exists at the replication site, the snapshot can be created on the table simply by making a small change in the script in Listing 2 (http://www.dbazine.com/code/Listing2.txt). Change Section 3 of the script as described in Listing 3 (http://www.dbazine.com/code/Listing3.txt). Notice the clause ON PREBUILT TABLE. This instructs Oracle that there is a table called TEST1 and that should be used as the segment for the snapshot named TEST1 and it should not create a new segment. The rest of the script in Listing 2 simply make the snapshot ready for replication.

After running the setup for awhile, the table TEST1 was altered as described above previously — the column COL3 was changed to CHAR(1000) and a new column COL4 CHAR(1) was added. These changes need to be reflected at the snapshot site, too. The recommended approach is to drop the snapshot at the snapshot site and run the process from beginning again; i.e., drop the table TEST1, create it, import rows, create snapshot on prebuilt table and finally generating replication support. With a large table, this may lead to several problems like running out of rollback segments, taking considerable time, and slowing down performance. In our case, it took more than four hours, including the table alters and, of course, the time will vary depending on your exact environment.

The Alternative Approach

When the snapshot is created on a table using the PREBUILT option, the snapshot simply takes over control of the segment defined for the table. When the snapshot is dropped, the segment is not dropped; rather it turns the control into the original table. Since the segment is the same, any data changes that occurred during the snapshot operation remain in the segment even after the snapshot is dropped. For example, say, the value of COL2 in the original table for COL1 = 1 was 'A.' After the snapshot creation on the table, the snapshot operation changed the data to 'B' since that was changed to 'B' at the master site. Subsequently, the snapshot was dropped, and the segment reverted to a table called TEST1. At this point, the value of COL2 for COL1 = 1 will still be 'B,' not 'A.' Thus, data in the segment remains exactly same, as it was the moment before the snapshot was dropped. This fact is exploited in the alternative approach.

Since data remains the same, there is no need to drop and rebuild the table from the master. We will use this trick to let the replication setup know that the snapshot was never dropped and so a fast refresh will work. Another detail to take care of at this time is the use of the already present snapshot log entries, which are needed for the fast refresh. When a snapshot is recreated on a prebuilt table, these entries on the master table are deleted. We will have to store them prior to the deletion and insert them after the snapshot is ready for replication.

Detailed Steps

- At the master site, the table can be altered using sql. Logging in as user ANANDA, issue the statements

```
alter table test1 add (col4 char(1);
alter table test1 modify (col3 char(1000));
```

- Since DML is still active on the table, you may have to wait until the table can be locked exclusively to add and modify the columns.
- At the replication site, we need to stop the replication pull for awhile by shutting down the job that does it. Logging in as user MVADMIN, issue

  ```
  select job from user_refresh where rname = 'TEST1';
  ```

- Note the job number. Again, as we assumed in the beginning, the refresh group is named TEST1. Shut down the job by issuing

  ```
  exec dbms_job.broken(<jobnumber>,TRUE);
  commit;
  ```

- It's very important to issue a commit here. You will also have to make sure no current sessions are currently active by this job. Check that by issuing

  ```
  select sid from dba_jobs_running where job = <jobnumber>;
  ```

- If you see any session, wait for it to finish or kill it before proceeding.
- Then, logging in as user ANANDA and issue

  ```
  DROP SNAPSHOT TEST1;
  ```

- This drops the snapshot but leaves the table in place. Then issue

  ```
  alter table test1 add (col4 char(1);
  alter table test1 modify (col3 char(1000));
  ```

The rest of the operation has been placed in a script as shown in Listing 4 (http://www.dbazine.com/code/Listing4.txt). Most of the script is the same as in the previous approach; the differences are explained here.

Section 3 is new. When a snapshot is built on an existing table and replication support is enabled on that, Oracle assumes that the snapshot has gone through a complete refresh; so it deletes the snapshot log entries at the master site. This is not acceptable in our situation, since the master table is undergoing some DML activity and generating snapshot log entries. Even if there is no DML, there could still be some unapplied snapshot log entries that must be preserved. Therefore, we must move the entries in the master table's snapshot log to a temporary table called mlog_bak. If the table name is too long, Oracle uses only the first 20 characters of the name to create the name of the snapshot log table. Although in this case the table name is only five letters, to make it generalized, we have used only the first 20 characters and prefixed it to *mlog$_* to get the name of the snapshot log table.

Sections 4 through 6 are the same as in the previous method.

- After the replication support is enabled on the table, the snapshot log entries preserved earlier need to be restored back so that they can aid in fast refresh. This is achieved in Section 7.
- After all these steps are executed issue a fast refresh of the snapshot just to make sure that the fast refresh works. Logging in as ANANDA issue

```
execute DBMS_snapshot.refresh('TEST1','F')
```

- This will do a fast refresh of the snapshot using the snapshot log entries we just restored back. The snapshot is all set for fast refresh and with the modified table structure.
- Finally, you have to re-enable the job broken earlier. Logging in as MVADMIN issue the following statement. Use the job number obtained before.

```
execute DBMS_job.run(<jobnumber>)
```

Total elapsed time, including the table alters, was 10 minutes.

Conclusion

As you can see the time to alter the master table was reduced from four hours to ten minutes, a 98 percent savings in time and more substantial savings in terms of effort for the DBA. Of course, your results may vary, depending on the table size and the network transfer speed. Nevertheless, no matter how fast the network speed is, unless the table is very tiny, the time and effort reduction will always be quite substantial.

Index