

Oracle7 Server Space Management

Revision 1.4b (95/10/31)

An Oracle Services Advanced Technologies Research Paper

ORACLE®

Oracle7 Server Space Management

An Oracle Services Advanced Technologies Research Paper

Part Number A00000-0 (no Oracle part number has been assigned)

Revision 1.4b (95/10/31)

Author: Cary V. Millsap

Contributing Authors: Carol Colrain, Dominic Delmolino, Craig Shallahamer, Vinay Srihari

Copyright © 1994, 1995 by Oracle Corporation. All rights reserved.

Printed in the U.S.A.

This document is distributed by Oracle Services as a supplement to the knowledge transfer process at specialty technical consulting visits.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free.

Oracle Corporation, 500 Oracle Parkway, Redwood Shores, California 94065.

The author or any other member of the Oracle Services Advanced Technologies team may be contacted via the Internet address: **atreq@us.oracle.com**. Please contact us directly if you would like to share your questions, comments, or observations.

Oracle, Oracle Server Manager, and SQL*Plus are registered trademarks of Oracle Corporation.

Oracle7, Optimal Flexible Architecture, and OFA are trademarks of Oracle Corporation.

All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

Contents

1. Our Goal.....	1
2. Data Architecture.....	1
2.1 Database and Database Files.....	1
2.2 Tablespace	1
2.3 Database Block.....	2
2.4 Extent and Segment	3
2.5 Storage Parameters.....	5
2.6 High-Water Mark.....	8
2.6.1 Measuring Blocks Allocated.....	10
2.6.2 Measuring Blocks With Rows.....	10
2.6.3 Measuring Blocks Above the High-Water Mark.....	11
2.6.4 Measuring Blocks Below the High-Water Mark	11
3. Sizing Tables, Clusters, and Indexes.....	12
3.1 Standard Sizes for All Extents.....	13
3.2 Extent Sizes for Multi-Block Reads	14
3.3 Minimum Extent Sizes.....	17
3.4 Scheduled Extent Allocations	18
3.5 The Maxextents Barrier.....	19
3.6 Space Conservation	20
3.7 Growth Monitoring and Adjustment.....	21
3.8 Summary of Recommendations	23
4. Sizing Rollback Segments	24
4.1 Standard Extent Sizes.....	25
4.2 Homogeneous Segment Sizes	25
4.3 Minimum Extent Sizes.....	26
4.4 Homogeneous Extent Sizes.....	26
4.5 Number of Extents	27
4.6 Number of Segments	29
4.7 Summary of Recommendations	33
5. Sizing Temporary Segments	33
5.1 Standard Extent Sizes.....	34
5.2 Extent Sizes for Multi-Block Reads	34
5.3 Minimum Extent Sizes.....	34

5.4 Homogeneous Extent Sizes	35
5.5 Synchronization with the Sort Area	36
5.6 Summary of Recommendations	38
6. Estimating Database Space Consumption	38
6.1 The Operating System Perspective	38
6.2 The Tablespace Perspective.....	39
6.3 The Database Block Perspective.....	39
6.4 The Row Perspective.....	40
6.5 Summary	40
7. Fragmentation.....	41
7.1 Tablespace Free Space Fragmentation.....	41
7.2 Block Fragmentation.....	44
7.2.1 Repairing Block Fragmentation	44
7.2.2 When to Truncate a Table	45
7.3 Row Fragmentation.....	47
7.4 Segment Fragmentation	47
7.4.1 The Mechanics of <i>exp</i> “Extent Compression”	48
7.4.2 Recursive SQL	50
7.4.3 Create and Drop	51
7.4.4 Drive Head Motion	51
7.4.5 Direct Access Paths	52
7.4.6 Real Benefits of Multiple Extents	53
8. Selected SQL Source Code	54
8.1 exts.sql — Extents Summary	54
8.2 se.sql — Segment Extent History	55
8.3 se.tbl — Create Objects for <i>se</i> Utilities.....	56
8.4 secol.sql — Collect Segment Extent History.....	56
8.5 sedel.sql — Delete Segment Extent History.....	58
8.6 spseg.sql — Storage Parameters for Segments	58
8.7 spts.sql — Storage Parameters for Tablespaces	60
8.8 tmap.sql — Tablespace Map by Block.....	61
8.9 tf.sql — Tablespace Free Space.....	62
8.10 tf.tbl — Create Objects for <i>tf</i> Utility	63
8.11 users.sql — User Privileges.....	63
9. References.....	64

1. Our Goal

The goal of the Oracle Server administrator is to allow application users to do their jobs quickly, without interruption, and with as little capital and labor investment as possible. For you to achieve this goal, you must comprehensively understand both how the Oracle Server manipulates space and how to use the tools Oracle gives you to control that process.

2. Data Architecture

The first step we'll take together in this document is to review the way Oracle7 stores and manipulates data. You may review the details of the concepts presented here in your standard Oracle Server reference material [1,2].

2.1 Database and Database Files

An Oracle *database* is a collection of operating system files. These files come in three flavors: *control files* define the relationship between physical Oracle files and the names that exist within the logical framework of the Oracle database dictionary; *redo log files* contain enough information to preserve the integrity of Oracle's transaction processing mechanism, in spite of the fact that database writing is asynchronous with respect to user requests to write database blocks; and *data files* contain the actual stuff that springs to mind when somebody says "database."

2.2 Tablespace

An Oracle database is partitioned into logical units of storage called *tablespaces*. A very simple Oracle database can consist of only one tablespace if you like (the one called **system**), but we recommend that you not operate a real Oracle database without at least five tablespaces:

1. **system**, to contain database dictionary segments only;
2. **rbs**, to contain rollback segments only;

3. **temp**, to contain temporary segments only;
4. **tools**, to contain segments for general-purpose tools like Oracle Forms; and
5. **users**, to contain users' segments.

When you create a table, cluster, index, or rollback segment in Oracle, you have the opportunity to specify exactly one tablespace within which you want that object to be allocated. Your configuration planner has made decisions about how to partition your database into different tablespaces to make administering your data much simpler. The *OFA Standard* describes Oracle's recommendations for deciding how to separate objects into separate tablespaces [4].

The tablespace is the "inside Oracle" object that defines contact with the physical data files that live in the "outside Oracle" world. A tablespace is made of one or more data files, which are specified when the tablespace is created with the **create tablespace** statement, and when a tablespace is given more space with the **alter tablespace ... add datafile** statement. When you ask Oracle to create and prepare an operating system file to become an Oracle data file, your server process formats it into Oracle *database blocks*.

2.3 Database Block

An Oracle database *block* is the physical unit of storage in which all Oracle database data are stored in the database files. The Oracle database block size is determined by the setting of the instance parameter called **db_block_size** when the database is created. A database configuration planner selects the block size to optimize the performance of the applications that will use that database. Many older Oracle databases use a 2 kilobyte database block size, because 2,048 is the traditional default database block size of Oracle software on most platforms. Benchmark testing and optimization of production systems worldwide have shown larger block sizes of 4, 8, 10, 16, or even 32 kilobytes¹ to improve performance of many applications on modern, memory-rich systems by reducing the number of database file read calls, especially for index depth traversal.

Because the block size of an Oracle database is a formatting definition, it cannot be changed simply by altering the value of **db_block_size** in the instance parameter file. If an Oracle Server administrator does make the decision to change the block size of a database, the only way to do so is to preserve the data (such as by using Oracle's **exp** utility [3]), rebuild the

¹ Consult your platform-specific Oracle Server documentation for details—not all of the database block sizes listed here are available on all ports.

database (by issuing the **create database** statement within Oracle Server Manager), and then repopulate the database (for example, by using Oracle's **imp** utility).

2.4 Extent and Segment

An *extent* is a contiguous allocation of database blocks. An extent is the allocation that is dedicated to any table, cluster, index, or rollback segment whenever (1) a segment is created or (2) a segment is required to grow because an insertion cannot be allocated into the existing allocation of space. A *segment* is a collection of extents that make up a single table, cluster, index, temporary segment, or rollback segment.²

Figure 1 shows a block-level description of a single tablespace named **users** in a database with **db_block_size** set to 2,048. You can obtain this picture for a given tablespace by printing the union of projections of the **dba_extents** and **dba_free_space** Oracle data dictionary views. The **tbmap** tool shown in section 8.6 will give you output similar to what you see here.

² The Oracle manuals will also tell you about bootstrap (or “cache”) segments, which are internal structures used behind the scenes by Oracle when you issue the **create database** statement. Aside from the fact that you'll see one in your **system** tablespace when you look, don't worry about them.

Tablespace	File#	Block#	Size	Type	Schema.Segment
users	19	1	1	file hdr	
		2	5	table	cary.a
		7	5	table	cary.a
		12	8	table	cary.a
		20	100	index	cary.a_u1
		120	5	free	
		125	5	free	
		130	8	free	
		138	375	free	
	20	1	1	file hdr	
		2	512	table	cary.a
		514	511	free	

Figure 1. This listing shows the relationship of a tablespace made of two data files, with the blocks that comprise it, and the extents that make up database segments. Each row in this figure not labeled “file hdr” or “free” represents an extent.

In this database, the **users** tablespace consists of two operating system files, which, although we cannot see their names in this picture, have internal Oracle file id numbers 19 and 20. The user named **cary** owns two segments in the **users** tablespace: one table named **a** and one index named **a_u1**. The table **a** consists of four extents, three in file 19 and one in file 20. The index **a_u1** consists of one extent that resides in file 19.

Note that a segment like table **a** can span operating system files by having some extents reside in one file and others in other files, but an individual extent must be a contiguous allocation of blocks. In this example, the user **cary** had encountered an ORA-01547 error, “failed to allocate extent of size 512 in tablespace ‘users’,” when some event motivated the attempted allocation of a 512-block extent to **a**, so he used the **alter tablespace** command to add a 2-megabyte data file to the **users** tablespace. That’s what motivated the server administrator to create file 20.

As you experiment, you might also note that your **dba_extents** and **dba_free_space** query will not naturally report on the one block of header present in block 1 of each Oracle data file (the **tmap** program shown in sec. 8.8 does some extra work to show block 1 in its report). Don’t confuse this block of file overhead with the segment header that occupies the first block of the first extent of every segment in the database. The first block in any Oracle

data file contains, for example, information written during instance checkpoints. The file header block is unusable by segments.

If you have ever tried to create a 1-megabyte table in a 1-megabyte tablespace, you now know exactly why your attempt failed. Some literature recommends making data files some “x percent” larger than a large extent that you will expect to reside there. You now know that to avoid wasting disk space, you want to make a data file exactly one Oracle database block larger than the amount of extent allocation you hope to store there. Some insightful configuration planners thus create tablespaces with sizes like **102402k** (100 megabytes plus one 2-kilobyte block) or **104859648** (the same thing expressed in bytes) instead of taking the simpler looking **100m** approach. That way they can use simpler storage parameters on their **create** statements later on.

2.5 Storage Parameters

A segment’s *storage parameters* are those values that define the initial allocation and growth rate of the segment. The creator of a segment chooses that segment’s storage parameters when creating the segment. For example, consider the following arbitrary table creation:

```
create table badt ...
tablespace users
storage (
  initial      5k
  next        26k
  pctincrease 10
  minextents  1
  maxextents  20
)
```

The table **badt** has storage parameters defined within parentheses after the **storage** keyword. The creator of a segment has enormous freedom in specification of storage parameters, including the freedom to specify no storage parameters at all. If no storage parameters are specified in a **create** statement, then the segment inherits the storage parameters defined in the **default storage** clause of the tablespace in which that segment will be created. The **spseg** (storage **p**arameters for **s**egments) tool shown in section 8.6 makes it easy to look at your storage parameters for your existing segments. For example,

```
SQL> @spseg gl gl_code_combinations%
```

will show you the storage parameters defined for all tables in the **gl** schema whose names match the pattern **gl_code_combinations%** (which, because of the naming convention used in Oracle Financials, will report on the code combinations table and all the indexes associated with that table). The **spts** report (storage **p**arameters for **t**ablespace) shown in sec-

tion 8.7 will show you the default storage parameters defined for each tablespace in your database.

The following segment storage parameters affect the size and number of extents allocated to a given segment:

- **initial** — the size of the segment's first extent.
- **next** — the size of the segment's next extent that will be allocated; the value of **next** is reset to the present value of **next** times $(1 + \text{pctincrease}/100)$ each time the segment gains a new extent.
- **pctincrease** — the percentage by which the $(k+1)$ th extent's size exceeds that of the k th extent.
- **minextents** — the number of extents allocated to the segment at creation.
- **maxextents** — the maximum number of extents that can be allocated to the segment.

To understand the details of how the **initial**, **next**, and **pctincrease** storage parameters affect the extent allocation for a segment, consider the table **t** that we created above with storage parameters (**initial 5k next 26k pctincrease 10 minextents 1 maxextents 20**). Let's assume that the database block size for our example is 2 kilobytes. Then creating the table would allocate as many whole blocks as necessary to provide at least 5 kilobytes of storage (**1** extent of size **5k**). So the initial extent created for **badt** would have three blocks, for a total size of 6 kilobytes.

Note that Oracle Server rounds up your extent size requests to the nearest block, so it is easier to calculate extent sizes in blocks rather than bytes. Remembering to round 2.5 blocks to 3 blocks is easier than remembering to round 5 kilobytes to 6 kilobytes in a database with 2-kilobyte blocks. If you try this experiment yourself on an Oracle7 database, you may also acquaint yourself with another phenomenon. Beginning with release 7.0 of Oracle, the server software uses a "gap-filling" storage allocation algorithm to prevent the creation of unusably small chunks of tablespace free space between segments. For example, if a tablespace had six blocks of free space lying between two extents, and a segment needed a new five-block extent, Oracle7 would allocate all six blocks to the new extent instead of allocating exactly the requested five blocks and leaving a one-block gap. Note that the **next** extent information stored in the Oracle7 data dictionary will thus not always exactly predict the size of the next extent allocation.

Extent	Extent Size in 2-Kilobyte Blocks	Extent Size in Kilobytes	Segment Size in Kilobytes
1	$\lceil 2.5 \rceil = 3$	6	6
2	$\lceil 13 \rceil = 13$	26	32
3	$\lceil 13 \times 1.10 \rceil = \lceil 14.3 \rceil = 15$	30	62
4	$\lceil 15 \times 1.10 \rceil = \lceil 16.5 \rceil = 17$	34	96
5	$\lceil 17 \times 1.10 \rceil = \lceil 18.7 \rceil = 19$	38	134
...
k	$\lceil \mathbf{next}_{k-1} \times (1 + \mathbf{pctincrease}/100) \rceil$		
...
20	$\lceil 85 \times 1.10 \rceil = \lceil 93.5 \rceil = 94$	188	1,646

Figure 2. As rows are inserted into a segment, Oracle Server allocates extents to that segment with sizes defined by the segment's storage parameters. This figure shows extent allocations for the **badt** table in a database with a 2,048-byte block size.

Let's now consider what happens to our table **badt** as we insert data. If we insert enough rows into **badt** that we need more space for data than the first 3-block, 6-kilobyte extent could hold, then Oracle Server will automatically attempt to allocate another extent to **badt**. The size of this extent will be 26 kilobytes, or exactly thirteen 2-kilobyte blocks, because that's what our setting of **next** requires. If row insertion continues to require extent allocations, the table **badt** will grow, until **badt** has up to 20 extents (because we chose **maxextents 20**), as shown in Figure 2.

Once all the data blocks of **badt**'s 20th extent are filled, any row insertion will fail with ORA-01631 "max # of extents (20) reached in table badt" because Oracle Server will not create more than **maxextents** extents for a segment. The maximum effective value of **maxextents** that you may use depends on the size of a single database block in versions prior to Oracle Server release 7.3; recall that the header block for any segment contains a map of all the extents associated with that segment. The number of entries that can be stored in that map is limited by the size of the block in which the map is stored. The maximum number of extents M that a segment can have is related to the **db_block_size** on most ports by the formula:

$$M = \frac{(b - \text{sizeof}(kcbh))/2 - \text{sizeof}(ktect)}{\text{sizeof}(ktetb)},$$

where b is the value of **db_block_size**, and $\text{sizeof}(c)$ can be found by querying the **v\$type_size** dynamic performance table:

```
select
  component, size
from
  v$type_size
where
  component = component
```

On a typical UNIX system, the **maxextents** formula reduces to:

$$M = \frac{(b - 24)/2 - 44}{8}.$$

The maximum number of extents allowed for each of several popular values of **db_block_size** is thus shown in Figure 3 for a typical UNIX system.

Value of db_block_size	Maximum Number of Extents per Segment
2,048	121
4,096	249
8,192	505
16,384	1,017
32,768	2,041

Figure 3. The maximum number of extents that an Oracle Server version 7.0, 7.1, or 7.2 segment can contain is a function of the value of **db_block_size** for the database on a given operating system. The figures shown here are for a typical UNIX implementation of Oracle Server. Version 7.3 is scheduled to remove the limit on the number of extents per segment.

2.6 High-Water Mark

The segment header block of a table or cluster also holds a value called the *high-water mark*. The Oracle7 high-water mark is similar in principal to the debris stains left on tree trunks

after a flood, which show the height of the water at its peak. The Oracle7 high-water mark obeys the following rules:

1. When a table or cluster is created, the high-water mark is set to the beginning of the segment.
2. The high-water mark is incremented in 5-block increments as rows are inserted into a table or cluster.
3. The high-water mark is decremented only by using the SQL **truncate** statement. The high-water mark does not recede when rows are deleted with the **delete** statement.

Oracle7 uses the high-water mark value to determine how many blocks to read when using the full-table (sequential) scan query access path. For example, imagine a table called **t**, initially allocated 512 kilobytes (256 blocks if we assume the use of 2-kilobyte blocks) of space with the following **create** statement:

```
create table t ...
tablespace users
storage (
  initial      512k
  next        512k
  pctincrease  0
  minextents  1
  maxextents  20
)
```

Let's say that we insert 2,000 rows, and that these rows fit into the first 20 blocks of the table. If we were to trace a full-table scan of these rows, we would see that Oracle Server has to read only about 20 blocks of data to fulfill our query. The server does not have to read all 256 allocated blocks because **t**'s high-water mark records the fact that no rows have ever existed beyond the 20th block in the table.

To understand the details of the Oracle7 high-water mark, let's examine another example, this time for a smaller table. Let's consider a small table with storage parameters (**initial 10k next 10k pctincrease 50**). In a test, we inserted 1,000 rows into this table, requiring 25 blocks of storage. The following table shows an example of the behavior of the allocation, the storage requirement, and the high-water mark.

Extent	Blocks Allocated	Blocks with Rows	Blocks Below High-Water Mark
1	5	4	4
2	5	5	5
3	10	10	10
4	15	6	10
Total	35	25	29

The following sections summarize how to make operational measurements on the high-water mark.

2.6.1 Measuring Blocks Allocated

The number of blocks allocated to a segment is reported in **dba_segments.blocks** (and also in **user_segments.blocks**):

```
select
  blocks
from
  dba_segments
where
  owner=upper(owner) and segment_name=upper(table)
```

2.6.2 Measuring Blocks With Rows

The number of table blocks that contain rows is reported in **dba_tables** (and **user_tables**) after using **analyze**:

```
analyze table owner.table estimate statistics
select
  blocks
from
  dba_tables
where
  owner=upper(owner) and table_name=upper(table)
```

An alternate method for finding the number of blocks that contain rows is to query the segment's **rowid** values:

```
select
  count(distinct substr(rowid,15,4) || substr(rowid,1,8))
from
  owner.table
```

You should experiment with both methods to determine which is better for your most interesting tables.

2.6.3 Measuring Blocks Above the High-Water Mark

You can find the exact number of blocks *above* the high-water mark for a given table by querying the **dba_tables** results produced by **analyze**:

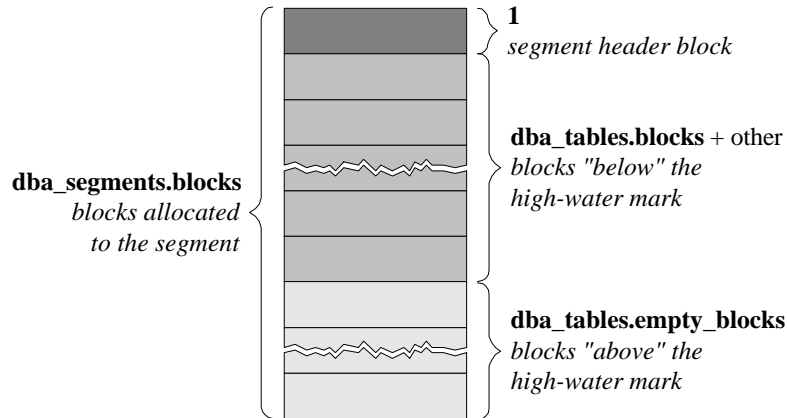
```
analyze table owner.table estimate statistics
select
  empty_blocks
from
  dba_tables
where
  owner=upper(owner) and table_name=upper(table)
```

2.6.4 Measuring Blocks Below the High-Water Mark

The number of blocks *below* the high-water mark is the most operationally useful statistic associated with the high-water mark. There is no direct method in Oracle7 for finding the number of blocks below the high-water mark. However, you can calculate this quantity by knowing the number of blocks allocated to the segment and the number of blocks *above* its high-water mark. If you have used **analyze** to populate or refresh the statistics shown in **dba_tables**, the number *h* of blocks below a segment's high-water mark is:

$$h = \text{dba_segments.blocks} - \text{dba_tables.empty_blocks} - 1.$$

This diagram shows the relationships of the values reported in **dba_tables** and **dba_segments**:



We have to plug the formula with the imprecise-looking component called “other” to account for two phenomena. First, blocks can be completely cleaned out by row deletions, which is recorded in **dba_tables.blocks**; but as we described earlier, the high-water mark does not recede except with the **truncate** command. You can in fact watch the value of **dba_tables.blocks** rise and fall as you insert and delete rows, but **empty_blocks** will never decrease unless you **truncate** the table. So the “other” category contains blocks that have been emptied but which still lie below the high-water mark. The “other” category also contains the number of blocks that have never actually contained rows but which are nonetheless below the high-water mark because of the five-block increment behavior of the high-water mark described above.

3. Sizing Tables, Clusters, and Indexes

The success of the Oracle Server administrator in meeting the space management goal of efficient reliability depends on that administrator’s expertise in choosing good storage parameters. In a nutshell, the strategy for achieving your Oracle space management goal is:

Set a growth schedule for each segment in your database, and choose storage parameters that will cause Oracle Server to allocate extents on that schedule.

For any segment that grows continuously, you should choose storage parameters that motivate Oracle Server to allocate a new extent to that segment about once every three to six months. In a 2-kilobyte block database, a schedule like this means that you can afford to run your system for about 30 years without having to rebuild the segment. With 4-kilobyte or larger database blocks, which nearly all production databases use today, a schedule that has each segment extent roughly once per quarter-year will support your database for over 60 years without segment rebuilds.

All of the segments created when your applications were installed have storage parameters that were defined when the segments were created. However, an application package is shipped to many different sites, each of which has different growth demands from all the others, so it is impossible for a single factory preset size to accurately accommodate the size requirements for all sites. The recommendations below form a sound space management strategy that will help you reach the space management goal of reliability and efficiency.

3.1 Standard Sizes for All Extents

Choose all **initial** and **next** storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.

The smaller the number of different extent sizes you use, the simpler your space administration task will be. Because dropping segments causes fragmentation of tablespace free space, you must choose your menu of extent sizes wisely. The list given above has the property that, if you only use extent sizes from this list, no matter what sized extent you drop, you can reuse all of the space you release without waste. Multiples of 2 kilobytes form the basis for the list because Oracle Server database block sizes are also multiples of 2 kilobytes.

In section 2.5 we described Oracle7's "gap-filling" storage allocation capability. This new Oracle feature was motivated by the fact that Oracle's default storage parameters³ yield extents of sizes (in blocks) 5, 5, 8, 12, 17, 26, 38, and so on. Before Oracle7's gap-filling capability, if you had dropped an 8-block extent that lay between two "live" extents, the only way to reuse the freed space would have been to allocate either another 8-block extent or to allocate a 5-block extent, which would have left 3 blocks free. The Oracle7 gap-filling capability eliminates some of the free space fragmentation trouble that once challenged Oracle

³ The default values for Oracle storage parameters are: **initial**, five database blocks; **next**, five database blocks; **pctincrease 50**; and **minextents 1**. The default value of **maxextents** varies by the setting of the **db_block_size** parameter.

Server administrators, but using standard extent sizes makes space management even simpler.

3.2 Extent Sizes for Multi-Block Reads

For segments that may be linearly scanned, set **initial** and **next** to some number $2^k b m$, where $k = 0, 1, 2, \dots$; b is the value of **db_block_size** for your database; and m is the value of **db_file_multiblock_read_count**.

Oracle Server administrators still debate whether or not allowing a segment to have more than one extent—“extent fragmentation”—is a harmful thing. A classic defense of the idea that extent fragmentation is bad is that having two or more extents associated with a segment prevents Oracle Server from using its efficient multi-block read capability. To consider the merit of this position, let’s quickly review what Oracle’s multi-block read capability is.

When an Oracle Server process is asked to perform a sequential scan of a table (a “full-table scan,” one of the most resource consumptive data access paths in Oracle), it uses a smart read-ahead algorithm to improve the response. The insights that produced this algorithm are (1) if we’re scanning a table from top to bottom, then we know exactly what blocks we’re going to need next, and (2) that the main cost of an I/O is the I/O setup, not the transfer of the contiguous data you pick up once you get there. It is important to remember that the Oracle Server can use its multi-block read capability *only* on full-table scans. When an Oracle Server process initiates a full-table scan, it reads n blocks from a data file, where n is the lesser of either the value of **db_file_multiblock_read_count** or the number of blocks remaining in the extent being examined.

The “demonstration” that extent fragmentation is bad then goes like this: Assume that a table has three extents, each with nine blocks, and assume that **db_file_multiblock_read_count** is set to 8. Then to full-scan the table will require two I/Os to read each extent, for a total of six I/Os, as shown in Figure 4. For each extent, one I/O will read eight blocks, and then a second I/O will read the remaining one block in the extent. If the table had been stored in one 27-block extent, then the full-scan would have required only four I/Os of 8, 8, 8, and 3 blocks, respectively.

That is an accurate observation and analysis, but there is an alternative recommendation. If we had created the table to have four 8-block extents, we could have achieved the same I/O efficiency result (four I/Os instead of six), as shown in Figure 5.

Extent fragmentation wasn’t the problem in this example, it was that the extents were sized not to mesh with the setting of **db_file_multiblock_read_count**. Even for small tables, careful selection of extent sizes removes the performance risk we’ve seen identified here.

Even the extent size becomes insignificant as tables get larger. For example, consider a 10-
extent table for which the extent size is 10,249 blocks, and the value of **db_file_multi-
block_read_count** is 32. Then each extent will consume 321 multi-block batches, the last of
which contains only 9 blocks. Now the difference between having ten extents and just one is
the difference between doing 3,210 I/Os or just 3,203—a penalty of only about a quarter of
one percent.⁴

So, for any access other than a full-table scan, multi-block reads are irrelevant to our deci-
sion making process. This means that for indexes, rollback segments, and tables that are
never read sequentially, there is no need to worry about extent sizing for multi-block I/O.
Our focus in this section then is on tables that *are* full-table scanned by an application (and
also on temporary segments, as recommended in sec. 5.2). For those segments, if you do
choose extent sizes that are integral values of **db_file_multiblock_read_count**, you put the
whole issue to rest. Conveniently, most administrators use **db_file_multiblock_read_count**
values like 8, 16, 32, 64, and so on (powers of two) that—guess what—work perfectly with
extents whose sizes are chosen from the set 2k, 4k, 8k, 16k, ..., as recommended in sec-
tion 3.1.

⁴ The arithmetic on this one is: $3,210 = \lceil 10,249/32 \rceil \times 10$; $3,203 = \lceil 102,49 \times 10/32 \rceil$;
and then $(3,210 - 3,203)/3,210 \times 100\% \approx 0.22\%$.

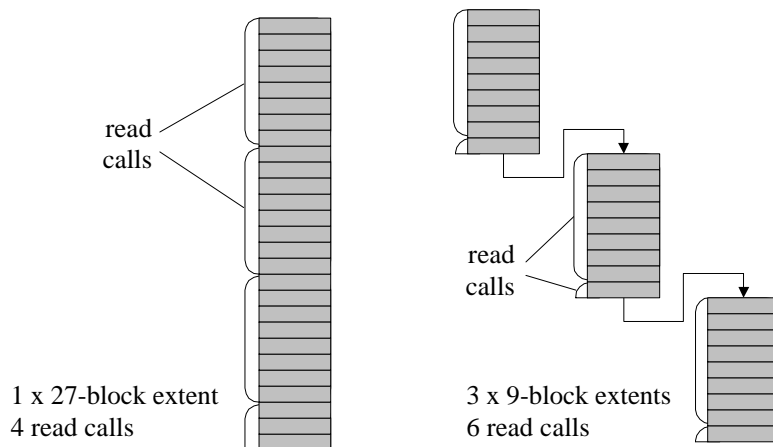


Figure 4. In the example above, `db_file_multiblock_read_count` is set to 8, but each extent of the multi-extent table contains 9 blocks, so two read calls are required to scan each extent—a total of 6 read calls required to scan the multi-extent table. To read the single-extent table requires only 4 read calls.

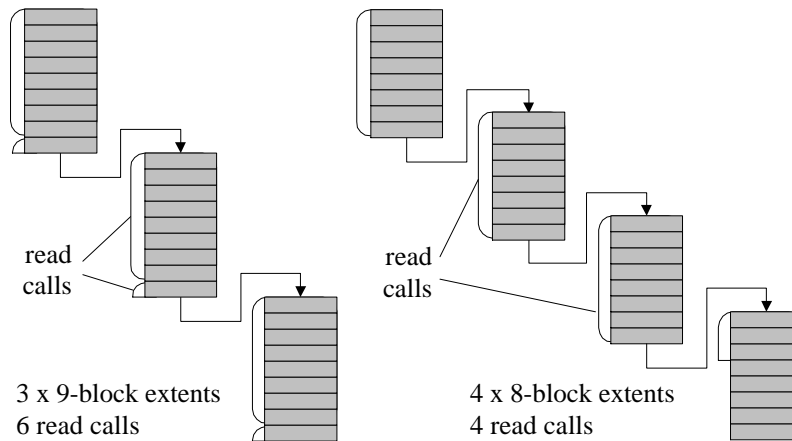


Figure 5. In this example, each extent of the multi-extent table contains 8 blocks, so there are four extents instead of just three. However, because only one read call is required for each extent, the four-extent table requires only four total read calls to scan the table. This is the same read call efficiency as the single-extent table shown in Figure 4.

3.3 Minimum Extent Sizes

Set **initial** and **next** for a segment to at least $1.25M/n$, where M is the predicted maximum size of the segment, and n is the maximum number of extents the segment can contain.

Selecting large enough extents ensures that you won't run out of extents when the segment grows up. The additional 0.25 is a hedge. For a segment that may eventually be M bytes large, you really want each extent to be at least M/n bytes large. Using $1.25M/n$ means that your estimate can be off by as much as 25 percent and you still will not run out of extents.

3.4 Scheduled Extent Allocations

Use **next = initial** and **pctincrease 0** for database segments that exhibit linear growth.

Almost all tables and indexes grow at a linear pace; that is, the rate of growth of almost any table or index can be expressed as a constant rate per unit of time. For example, a server administrator might quantify that a General Ledger balances table grows at the rate of 20 megabytes per month. This is a linear growth rate. Even if the administrator's company acquired another business and grew the balances table at the increased rate of 30 megabytes per month, then the table still has a linear, albeit new, rate of growth. Using **next = initial** and **pctincrease 0** will ensure that all the extents associated with that segment will be the same size, which will make your space management task much simpler.

Some data do grow non-linearly, but usually not for long before nature's laws of finite resource constraints takes over. Using the default **pctincrease 50** for a segment is appropriate only if the data in that segment grow 50 percent more quickly each day. Most databases don't do that, so using **pctincrease 50** is almost never the best choice you can make.

Administering extent allocation is much easier if (1) it is easy to predict the size of the next extent that will be allocated and (2) that next extent will be no larger than the last extent that was allocated. For example, consider a tablespace with 300 megabytes of free space, and a table that, by growing 5 megabytes per month, will eventually require a steady-state 250 megabytes of that space.⁵ Selecting storage parameters (**initial 10m next 10m pctincrease 50**) would require the following allocations to fulfill the 250-megabyte storage requirement:

⁵ The reason we can claim that a table will reach "steady-state" at a particular limit is that most applications allow the server administrator to archive and purge old rows at approximately the same pace as new rows come in. For example, a general ledger's account balance rows more than three years old are usually archived to tape and removed from the database. In this way, this year's data are able to reuse space once dedicated to storage of four year-old data.

Extent	Extent Size in Megabytes	Cumulative Table Size in Megabytes
1	10.00	10.00
2	10.00	20.00
3	15.00	35.00
4	22.50	57.50
5	33.75	91.25
6	50.63	141.88
7	75.94	217.81
8	113.91	331.72

There are a couple of things wrong with this picture. First, we will run out of tablespace free space before we can allocate the 8th extent—remember, there were only 300 megabytes of free space available. Second, we needed only 250 megabytes to fulfill our requirement, yet with these storage parameters, Oracle Server will have to allocate almost 332 megabytes to accommodate the table. Using (**initial 25m next 25m pctincrease 0**) would accomplish our goal of storing the 250-megabyte table without exceeding the capacity of its host tablespace, and in fact without any appreciable waste of space. Using **pctincrease 0** in this case makes the administrator’s management task much simpler. The first piece of evidence to support this conjecture is that you don’t need a picture to see how the table would grow using the second set of storage parameters. The table would require simply 10 extents at 25 megabytes apiece.

3.5 The Maxextents Barrier

Set **maxextents** to a value that is smaller than the enforced maximum number of extents that the segment is allowed, and monitor segments whose extent allocations approach the **maxextents** barrier. If a segment nears its effective **maxextents**, then alter its **next** value to stave off application failure, and then rebuild the segment with appropriately sized extents at your next opportunity.

Many developers and server administrators choose **maxextents** values less than the allowed maximum as a safety valve. If an unexpectedly fast-growing segment escapes the attention of the administrator, and an application fails with a “max # of extents reached...” error, the problem can be repaired by altering the storage parameters of the segment, without having to rebuild the segment. For example, if our table **badt** that we discussed in section 2.5 were to hit its 20-extent barrier, causing an ORA-01631 error, then we could choose a more appro-

appropriate next storage parameter value to prevent the table from extending so often, and we could increase the value of `maxextents` to allow for some more growth of **badt**, as in:

```
alter table badt
storage (
  next          1m
  pctincrease   0
  maxextents    30
)
```

If `maxextents` for **badt** in our 2-kilobyte block database had been set to 121, then our only option for moving beyond the ORA-01631 error would have been to rebuild **badt** with more appropriate storage parameters. The ensuing preserve-the-data, **drop**, **create**, and restore-the-data cycle would have been enormously more expensive than the **alter table** solution.

The expert Oracle Server administrator will never intentionally let an ORA-xxxxx error afflict an application, but having some slack in your `maxextents` settings will give you the chance to recover gracefully in case a rapidly growing segment somehow escapes your attention.⁶

3.6 Space Conservation

For static (fixed-size) segments, use multiple extents whenever necessary to avoid wasting more than about 10 percent of the space allocated to the segment.

Many administrators want (1) never to waste space and (2) always to ensure that no segment uses more than one extent. Now we're recommending that you (3) choose extent sizes only from the list described in section 3.1. We would be denied success if we had to meet *all* of these constraints because they cannot be met simultaneously. Fortunately, the belief that application performance will suffer if a segment consists of more than one extent is not true, so we can relax the second constraint.

Consider a 1.1-megabyte table that contains application help text. The size of the help text table is constant; it will change only if the application is replaced (e.g., upgraded). If we insist on creating the table in one extent, then we have to choose **initial 2m** if we are going to follow our own recommendation on extent sizing. If we relax the constraint that says all segments should contain no more than one extent, then we can pick (**initial 128k next 128k pctincrease 0**) as storage parameters for the help table, requiring the table to consume only

⁶ The **se** family of programs given in section 8 will help ensure that no such event will escape your attention.

five extents, and with practically no storage waste. Recall two facts to ease your mind about whether we've introduced an inefficiency by using more than one extent:

1. A well-designed application will access its help text table by a direct-access (index- or hash-based) execution plan. The multi-block read argument proposed by the advocates of always using one extent per segment cannot be applied to segments that are not sequentially scanned.
2. If the application help text table *is* ever sequentially scanned, then the extent sizes recommended in this paper will exploit the multi-block read capability of Oracle Server every bit as well as if we had used a large single extent.

3.7 Growth Monitoring and Adjustment

Carefully monitor segment growth patterns. If a segment's growth pattern changes, then alter the segment's **next** or **pctincrease** values appropriately.

Fear is probably the number one enemy of effective space management at sites using pre-packaged Oracle applications. Especially at sites whose server administration staff are astonished in a project's first several weeks with the amount of labor required to plan, configure and install a complicated application, the temptation to leave things alone can be overwhelming. However, no vendor can accurately predict the volume of data that you will pump into your Oracle database. Successful space management depends on your ability to model the growth of your real-world data with your Oracle storage parameters. Since your data growth is unique to your site, your storage parameter settings will be unique to your site also.

Data growth is seldom perfectly uniform, and you need a flexible space management procedure that allows you to mid-course correct for the inevitable changes in character that your data growth will take. To accomplish this, you need more information than just how many blocks are in use in an extent. You need to know the actual *rate* of extent allocation to each segment. While the Oracle Server does maintain a history of extent allocation for every segment in the **dba_extents** view's base table, no data are collected on the actual times of allocation.

If timing information on extent allocation were captured, you would be able to predict the growth rate of a segment down to per-extent granularity. You would then be able to define next extent sizes so that your segments would extend at regular time intervals. An ability to predict and control your database growth would allow you to make accurate requests for hardware capacity. (Your operating system administration friends may highly appreciate this kind of accuracy.)

Consider the following four-extent segment (assume it's a table):

Extent	File	Block	Size
1	1	231	25
2	3	10	25
3	2	9,785	25
4	4	504	25

There are several ways that a table could exhibit this kind of extent allocation. The table may be a static, query-only table that has been deliberately striped across files within a tablespace to speed up common queries. The extents may have been deliberately pre-allocated in anticipation of a particular growth rate. Or the table may have grown this way over time on its own, using its defined storage parameters. Without timing information, one can only guess about the storage plan for this table.

Now consider the same extent allocation with timing information:

Extent	File	Block	Size	Detection Time
1	1	231	25	93/10/01 23:25
2	3	10	25	93/10/01 23:40
3	2	9,785	25	93/11/01 23:38
4	4	504	25	93/11/01 23:54

It now becomes obvious that this table exhibits a regular growth pattern that can be easily anticipated. If today's date is November 10, 1993, we know that the table extends twice a month on the same day (perhaps during a batch load), by a total of 50 blocks per month. You have accurate information now that will enable you to make an informed estimate of when the table will extend again.

Since you now have size and time information, you can calculate growth rate by dividing the sum of extent sizes by the amount of time it took to allocate those extents (100 blocks divided by two months yields a rate of 50 blocks per month). If you had this kind of information for every table in your database, you would be able to formulate a query that tells you how to define **next** extent size information for every table such that all your tables will extend once in a given time period; e.g., "I'd like all my tables to extend once per quarter," "...per year," and so forth. Sections 8.3 through 8.5 contain sample SQL programs to perform this statistical gathering and query formulation, as in the following example:

```

SQL> connect sys          connection to sys is required to run se.tbl
SQL> @se.tbl              prepare the database for se utilities
SQL> connect dom          dom is a schema with DBA privileges
SQL> @secol % %           execute regularly (e.g., each night) in an automated batch
SQL> @se gl %             e.g., to view gl schema extent history

```

3.8 Summary of Recommendations

The following list is a summary of the space management techniques we have recommended for tables and indexes:

1. Choose **initial** and **next** storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.
2. For segments that may be linearly scanned, set **initial** and **next** to some number $2^k bm$, where $k = 0, 1, 2, \dots$; b is the value of **db_block_size** for your database; and m is the value of **db_file_multiblock_read_count**.
3. Set **initial** and **next** for a segment to at least $1.25M/n$, where M is the predicted maximum size of the segment, and n is the maximum number of extents the segment can contain.
4. Use **next = initial** and **pctincrease 0** for database segments that exhibit linear growth.
5. Set **maxextents** to a value that is smaller than the enforced maximum number of extents that the segment is allowed, and monitor segments whose extent allocations approach the **maxextents** barrier. If a segment nears its effective **maxextents**, then alter its **next** value to stave off application failure, and then rebuild the segment with appropriately sized extents at the next opportunity.
6. For static (fixed-size) segments, use multiple extents whenever necessary to avoid wasting more than about 10 percent of the space allocated to the segment.
7. Carefully monitor segment growth patterns. If a segment's growth pattern changes, then alter the segment's **next** or **pctincrease** values appropriately.

4. Sizing Rollback Segments

A *rollback segment* is a structure that Oracle Server uses to store transaction *undo*. Undo is required whenever a server process fulfills a **rollback** request,⁷ or when a server process needs to construct a self-consistent point-in-time image of some data being queried. To help you see why undo is important for read consistency, let's consider an example. Say that at 12:00 noon, the user **cary** begins a query of table **a**, and that **a** is so large that the query will not finish until 1:00 P.M. At 12:15, the user **craig** commits an update that modifies the first and last rows of **a**. By 12:15, **cary**'s query has already printed out the first row of **a**, so it's easy to see that the output already on paper shows the table as it existed at 12 noon. But by the time the query encounters the last row of **a** at 1:00, that row no longer looks as it did at 12:00, because **craig** has changed it.

Time	Event
12:00 NOON	cary 's query begins and prints contents of table a block 1
12:15 P.M.	craig 's update changes and commits table a blocks 1 and 1,000,000
1:00 P.M.	cary 's query prints contents of table a block 1,000,000

So Oracle could have been designed any of three ways:

1. Oracle could have printed a mutated result in which some parts of the printout reflect the way the table looked at 12:00, and other parts reflect how the table looked at 1:00—clearly not an acceptable result in many cases (for example, think about a query that compares salaries in a payroll table while in the midst of a company-wide transaction that implements raises).
2. Oracle could have simply chosen to return an error message saying, “Sorry, this table is being updated, no reads allowed”—which would have been of zero use in any realistic environment.

⁷ Such a rollback request can be made either explicitly, such as when a user process executes a **rollback** statement, or implicitly when a server process rolls back uncommitted transactions during instance recovery.

3. Or Oracle could have chosen to record the way the data looked at 12:00 noon, so that a query that began before 12:15 could print the 12:00 rows, even if they've changed since the query began.

Option 3 is how Oracle works. This 12:00 record is stored as undo in a rollback segment.

Rollback segment blocks page in and out of the Oracle SGA just like the blocks of any other Oracle segment. Physically, rollback segments are collections of extents just like tables and indexes; and, like tables and indexes, rollback segment growth is governed by the normal laws of extent allocation using **initial**, **next**, **minextents**, and **maxextents**. However, because rollback segments are logically regarded as a circular queue of extents, they are required to have a **minextents** value of at least 2; and Oracle7 rollback segments always have an **pctincrease** value of 0—you may not specify a **pctincrease** value.

Rollback segments, unlike any other kind of Oracle7 database segment, can shrink as well as grow. Rollback segment shrinkage is governed by a special storage parameter, called **optimal**, which defines the smallest size beyond which the rollback segment will not shrink.

4.1 Standard Extent Sizes

Choose all **initial** and **next** storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.

Since rollback segment extents are frequently de-allocated and re-allocated as a matter of course (because Oracle7 automatically shrinks and grows rollback segments within the confines of their storage parameters), it is especially important to select extent sizes from a small, well-chosen list. Please see section 3.1 for a full explanation of why we recommend the particular list shown here.

4.2 Homogeneous Segment Sizes

Make all your rollback segments the same size.

A transaction chooses which rollback segment to use for its undo by using a modified round-robin algorithm. It is difficult both (1) to forecast which rollback segment will be obtained by a given transaction on a busy system, and (2) to forecast how many transactions will be active in a given rollback segment at an arbitrary time. On a busy system with a mixture of large and small transactions, Oracle7 rollback segments will stretch to accommodate the large transactions, and they will shrink to prevent space waste when they can. Consequently, a rollback segment's size over time tends toward the size of the undo generated by the most

frequently recurring transactions, limited at the low end by the setting of **optimal** and at the high end by the setting of **maxextents**.

Because an administrator is generally unable to predict which rollback segment will be chosen for a given transaction, all rollback segments tend to become the same size over time under consistent transaction load. We recommend that the administrator pre-create all rollback segments to be the same size using the same storage parameters.

4.3 Minimum Extent Sizes

Set **initial** and **next** for each rollback segment to at least $1.25M/n$, where M is the predicted maximum size of the undo written to a single rollback segment concurrently by your application, and n is the maximum number of extents a segment can contain.

If a transaction generates more undo than a rollback segment is ultimately able to hold, then that transaction will fail. Your large transactions dictate the ultimate size of your rollback segments, and since all Oracle segments' sizes are limited by the number of extents your database block size allows you to have, your large transactions dictate the size of the extents in your rollback segments.

Note that it is possible for two or more transactions to generate undo to the same rollback segment concurrently. Thus, it is possible that a single rollback segment will need to grow beyond the size of the largest transaction on your system. You can control this phenomenon by using the **set transaction use rollback segment** statement and by ensuring that there are sufficiently many rollback segments on line to reduce the possibility that two large transactions will generate undo to the same rollback segment (sec. 4.5).

If n is the number of extents that your choice of **db_block_size** will allow your rollback segment to have (sec. 2.5), then using extent sizes that are at least $(1/n)$ th as large as your rollback segment will ever get will ensure that your rollback segments will never run out of extents. Because forecasts are rarely perfect, we recommend that you make your extent sizes $1.25/n$ times the size of your forecast maximum rollback segment size. The additional 0.25 will protect you in case your estimate is up to 25 percent short.

4.4 Homogeneous Extent Sizes

Use **next = initial** for all rollback segments.

Because of the way Oracle Server regards rollback segments as rings of extents, varying the extent sizes of a rollback segment complicates the job of forecasting rollback segment behav-

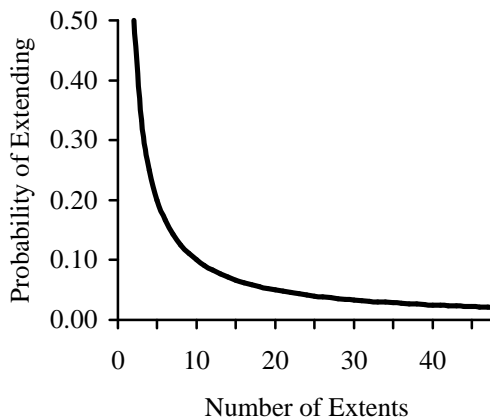


Figure 6. The probability that a rollback segment will require a new extent is inversely related to the number of extents in that rollback segment. Beyond 20 extents, the incremental decrease in the segment’s probability of extending fails to justify the cost of an additional extent.

A site with large rollback segments capable of handling enormous volume per single transaction can fall into the trap of thinking that extent fragmentation is bad and build, for example, 100-megabyte rollback segments that each contains only two extents. The problem with this decision is that, for example, a single 12-byte uncommitted transaction residing in the tail-end of extent 1 will require Oracle Server to allocate a third extent for the rollback segment when the undo write pointer is ready to switch from extent 2 into extent 1, and—presto—the administrator is the owner of a three-extent, 150-megabyte rollback segment, and there didn’t even have to be a large transaction running to make it happen. There is also a performance penalty associated with the execution of recursive SQL required to make the dynamic extension of the rollback segment happen.⁸

The probability that a rollback segment will require a new extent allocation thus depends upon how likely the next extent is to contain active undo. Assume that a given rollback segment has a single tiny amount of active undo from another transaction stored in it some-

⁸ Most popular Oracle Server tuning texts recommend that to reduce recursive SQL, the server administrator should try to minimize the number of extents in each Oracle segment. Ironically, with rollback segments, having a smaller number of extents actually induces *more* recursive SQL, which harms performance.

where, and that the undo write pointer for that rollback segment has filled its current extent and seeks to enter the next extent in the ring. If that rollback segment has only two extents, then the probability that the next extent in the ring contains the active undo for that other transaction is 1 in 2, or 0.50. If that rollback were to have three extents, then the probability of requiring a new extent allocation drops to 1 in 3, or 0.33. The more extents that the rollback segment has, the less likely it is that the rollback segment will require an extent allocation that could have been avoided, as shown in Figure 6.

So it is in the administrator's best interest to create rollback segments with many extents. Naturally, there is a point of diminishing returns. Figure 6 shows the rapid decline of a rollback segment's probability of extending as a function of increasing numbers of extents. The curve flattens out sufficiently at 20 extents that the number 20 becomes our specific recommendation as the number of extents you initially allocate to your rollback segments.

The only cost of having more extents is the risk of encroaching the segment's effective **maxextents** value if the segment were to grow unexpectedly. Beyond 20, the incremental decrease in the segment's probability of extending fails to justify the cost of an additional extent. However, with an initial allocation of 20 extents, there's plenty of space left over in the rollback segment header's extent map to accommodate a lot more extent allocations, even if you use only a 2,048-byte (small) database block size.

There is one more important point worth mentioning about the number of extents that a rollback segment should contain. Please don't fall prey to the fallacy that you need to have a small number of extents in your rollback segments so that your rollback segments will be more likely to remain cached in your SGA. Remember, Oracle doesn't manipulate *extents* in the database buffer cache, it manipulates *blocks*. An *n*-block rollback segment with one hundred extents is exactly as likely to remain cached in the database buffer cache as an *n*-block rollback segment with two extents.

4.6 Number of Segments

Create enough rollback segments to prevent undo header waits, but do not create more rollback segments than your instance's maximum number of concurrently active transactions.

Every transaction requires update access to a data structure called a *transaction table*, which resides physically in the header block of the rollback segment chosen to hold the undo for the transaction. When two transactions assigned to the same rollback segment simultaneously attempt to update the same header block, one of the two transactions will endure an *undo header wait*. Naturally, you don't want processes to have to wait on each other in your DBMS, so you would like to have enough rollback segments available to reduce the possibil-

ity of two transactions' needing to update the same rollback segment header block at the same time.

However, we've already suggested that you make all your rollback segments the same size, and that the largest recurring transaction on your system is the one that determines what that size should be. It is not unusual if your site's transaction character goes something like the following:

Attribute	Quantity
named users	300
concurrently active interactive users	170
concurrently active batch programs	10

The big batch jobs are going to require big rollback segments—let's say, for example, of 100 megabytes apiece. But there are 170 users in addition to those batch connections executing transactions that can require simultaneous update access to rollback segment headers.

There is no way that a site with this kind of user load should reasonably expect to devote $(125 + 10) \times 100$ megabytes of disk space to rollback segments—that's 13.5 gigabytes. The actual number of rollback segments required to eliminate undo header waits is one for each concurrently active transaction. The important observation here is that “per concurrently active *transaction*” is vastly different from “per concurrently active *user*” on most systems with an interactive/batch user mix. Here's why.

Most users interact with the database through forms-based applications that require a user to complete a screenful of information before the form fires off the **insert**, **update**, or **delete** statement, which is then immediately followed by a **commit**. Oracle Server views a transaction as beginning only when the **insert/update/delete** statement arrives for execution and ending when the **commit** takes place. Transactions in a forms-based interactive environment typically have a very short, bursty nature. Assume that, at peak interactive load on our 170-user system, each user executes 3 transactions per minute (a commit every 20 seconds), and that the average transaction duration is 1 second. For the moment, also assume that there is nothing running in batch. If we choose an arbitrary time during the day, we can ask questions about the likelihood that any two transactions will be simultaneously active, any three, and so on, to determine how many rollback segments we will theoretically need to prevent undo header waits.

If you have had some exposure to elementary probability theory, what we've discussed so far may look a lot like a story problem you've seen before. In fact, if our assumptions really do hold (170 users at three commits per minute, and one-second transaction duration), then we can *calculate* the probability of there being a specific number of transactions active at any

arbitrarily chosen instant, using the binomial distribution formula. The probability that exactly x transactions will be active at an arbitrary moment is:

$$P(X = x) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}, \quad x = 0, 1, 2, \dots, n$$

where n is the number of users on the system (170), and p is the probability that any one transaction will be active at an arbitrarily chosen moment (3 transaction-seconds per 60 elapsed-seconds = 0.05). If we graph this function for the entire range of interesting values of x , we get the bell curve shown in Figure 7.

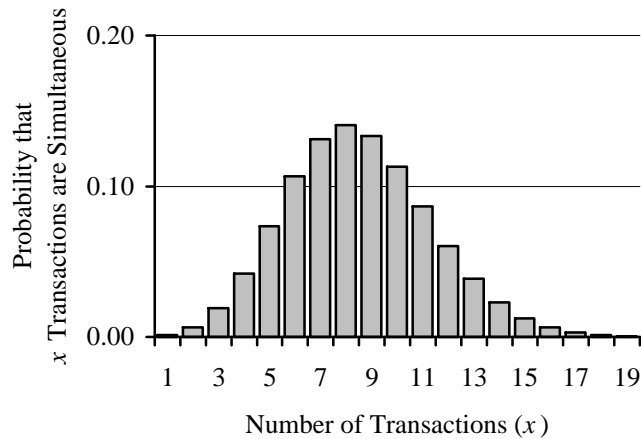


Figure 7. The probability that x transactions are active at a given instant defines a bell curve. In this example, $n = 170$ users, and $p = 0.05$ (3 one-second transactions each 60 seconds). The probability that at a given instant there will be more than more than 18 simultaneously active transactions on this system is $P(X > 18) = 1 - P(X \leq 18) \leq 0.001$.

In words, what we see is that on a 170-user system with users executing three one-second transactions per minute, at any arbitrarily chosen moment there will probably be between about four and thirteen simultaneously active transactions on this system. There is a 50 percent chance that at any given time, there will be at least eight concurrently active transactions. However, the probability that there will be more than eighteen concurrently active transactions on the system at a given time is less than one tenth of one percent. So we've learned that, if interactive processes are the only things on this system, and if our as-

sumptions hold about the character of those interactive transactions, then we will practically eliminate undo header waits on our system if we create only 18 rollback segments.

Recall at this point that the *Oracle Database Administrator's Guide* recommends use of one rollback segment for about every four concurrent transactions [2]. The model we have generated thus far has assisted in predicting only how many simultaneously active, concurrent transactions we can expect at a given point in time. Not every phase of an Oracle “write undo” operation requires access to the rollback segment header. Consequently, odds are in your favor that you can get by with even fewer than the number of rollback segments for interactive processes than is predicted here. Although we could not recommend creating only five rollback segments for this interactive load, the right answer almost certainly lies between 5 and 18.

But we're not done yet—what about our batch processing requirements? We could pop out another fancy analysis, this time for batch programs, but we would end up learning that the number of batch transactions we expect to be active at any given moment is exactly the number of batch programs that we allow to run simultaneously, because the probability p of a transaction's being active during a long-running, update-intensive batch process is nearly 1. For example, if we're allowed to run ten batch updates at the same time, and if the queues are all busy most of the time, then there is an excellent chance that there will be ten concurrently active transactions in batch sometime during the day. If this were indeed the case in the example we're working on, then the maximum number of rollback segments we should create is 28—that's 18 to accommodate interactive transaction processing, and 10 more to accommodate the batch transaction processing.⁹

Our simple predictive model can help you reasonably estimate the number of rollback segments you might need. Being well-informed when you select your starting point makes it easy for you to achieve your real goal—to eliminate undo header waits—by using the Oracle Server Manager **monitor rollback** tool or the **v\$waitstat** dynamic performance view. If you see undo header waits and you have enough disk space, then add another rollback segment. If you don't see undo header waits, then you might try dropping a rollback segment to reclaim some disk space. If removing a rollback segment causes your system to incur undo header waits, then add it back.

⁹ Please remember that the recommendation made here is based on the assumptions that users execute a consistent three transactions per minute, and that transaction duration is one second. These assumptions are considered only to lend concreteness to our example—they are *not* based on tests and measurements taken on actual forms-based interactive systems. Compute these numbers for your system using a SQL trace analysis and a stopwatch.

4.7 Summary of Recommendations

The following list is a summary of the space management techniques we have recommended for rollback segments:

1. Choose all **initial** and **next** storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.
2. Make all your rollback segments the same size.
3. Set **initial** and **next** for each rollback segment to at least $1.25M/n$, where M is the predicted maximum size of the undo written to a single rollback segment concurrently by your application, and n is the maximum number of extents a segment can contain.
4. Use **next** = **initial** for all rollback segments.
5. Set **minextents 20** for each rollback segment, and set **optimal** to the appropriate value to ensure that rollback segments do not shrink below 20 extents.
6. Create enough rollback segments to prevent undo header waits, but do not create more rollback segments than your instance's maximum number of concurrently active transactions.

5. Sizing Temporary Segments

When an Oracle server process performs a sort that will not fit into the sort area (whose size in bytes is defined as the value of the **sort_area_size** instance parameter), the process writes sorted intermediate results to a *temporary segment* in the database. Temporary segments also consist of extents—just like tables, indexes, and rollback segments—and they obey the same physical laws of growth that tables and indexes obey, using the same kinds of storage parameters. A key difference of temporary segment creation is that it is always implicit. A user never explicitly requests creation of a temporary segment (there is no “create temporary segment” statement), so the user never has the chance to specify where the segment should be created, or what storage parameters to use.

The tablespace defined as a user's *temporary tablespace* is the one in which the user's temporary segments will be created. The storage parameters of a temporary segment are those defined as the default storage parameters for that tablespace. The Oracle Server administrator can observe the users' temporary tablespaces by running a report like the **users** program shown in section 8.10. You have already seen that you can look at a tablespace's default storage parameters with the **spts** report. It is critical that you never allow a user to have its

temporary tablespace set to **system**. You must pay attention to the default storage parameters of any tablespace that shows up in the **temporary_tablespace** column of the **dba_users** data dictionary view.

5.1 Standard Extent Sizes

For tablespaces that are used for holding temporary segments, choose **initial** and **next** default storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.

This recommendation may seem familiar to you by now, but note that we have phrased the standard extent sizes recommendation a bit differently this time. Since storage parameters cannot be specified explicitly when a temporary segment is created, the server administrator must take care to set the **default storage** parameters for any tablespace that acts in the capacity of temporary tablespace for any user. See section 3.1 for details about why we recommend these particular powers-of-two extent sizes.

5.2 Extent Sizes for Multi-Block Reads

For tablespaces that are used for holding temporary segments, set **initial** and **next** default storage parameters to some number $2^k bm$, where $k = 0, 1, 2, \dots$; b is the value of **db_block_size** for your database; and m is the value of **db_file_multiblock_read_count**.

We have already seen that if we choose extent sizes that don't match the value of **db_file_multiblock_read_count** for your instance, then we can incur more I/O calls than necessary during full-scans. Temporary segments are indeed accessed sequentially, so it is good to choose extent sizes that are some integral multiple of the multi-block read batch size. Thus, if your **db_file_multiblock_read_count** is set to 32, and your **db_block_size** is 2,048, then it is a good idea to choose **initial** and **next** default storage parameters for temporary tablespaces from the list 65,536 ($k = 0$), 131,072 ($k = 1$), 262,144 ($k = 2$), and so on. See section 3.2 for full details.

5.3 Minimum Extent Sizes

For tablespaces that are used for holding temporary segments, set **initial** and **next** to at least $1.25M/n$, where M is the predicted maximum size of your largest sort to disk, and n is the maximum number of extents a temporary segment can contain.

Your large sort operations dictate the size of your temporary segment extents: a temporary segment must be able to acquire enough extents to hold the sort runs it will be required to keep. For you to be able to choose extent sizes for the tablespaces used for temporary segments, you have to be able to forecast the size of the largest sort to disk on your system. This forecast can be difficult to make, because you don't have explicit control over sorting; you don't even have explicit control over when a sort will occur.

The easiest way to forecast the amount of temporary space you'll need is to pick the most demanding process on your system that uses a **create index, order by, distinct, group by, union, intersect, or minus**. (You probably have a pretty good idea of what that is: if you're an Oracle Financials site, it's probably a gigantic General Ledger posting batch; if you're an Oracle Manufacturing site, it's probably your nightly MRP run; and so on.) Run this process in a test environment with a fresh temporary tablespace that has no "honeycomb" tablespace free space fragmentation (sec. 7.1). Then when the process is complete, check your **tbmap** report (sec. 8.6) for the temporary tablespace to see the total allocation that was made for that process. This will give you an idea about how large the sort was.

Once you figure out how much temporary space your most challenging process consumes, you can choose extent sizes appropriately. As we described in sections 3.3 and 4.3, the recommendation listed here allows you to overshoot your forecast by up to 25 percent without incurring an application failure.

5.4 Homogeneous Extent Sizes

Choose default storage parameters for all tablespaces that will hold temporary segments so that **next** is some integral multiple of **initial**, and use **pctincrease 0**.

There is no need for one temporary segment allocation to be geometrically larger than the prior allocation for that segment, so we recommend using default storage parameters on tablespaces that hold temporary segments so that **next = initial**. Further details on the general topic of extent sizes can be found in section 3.4.

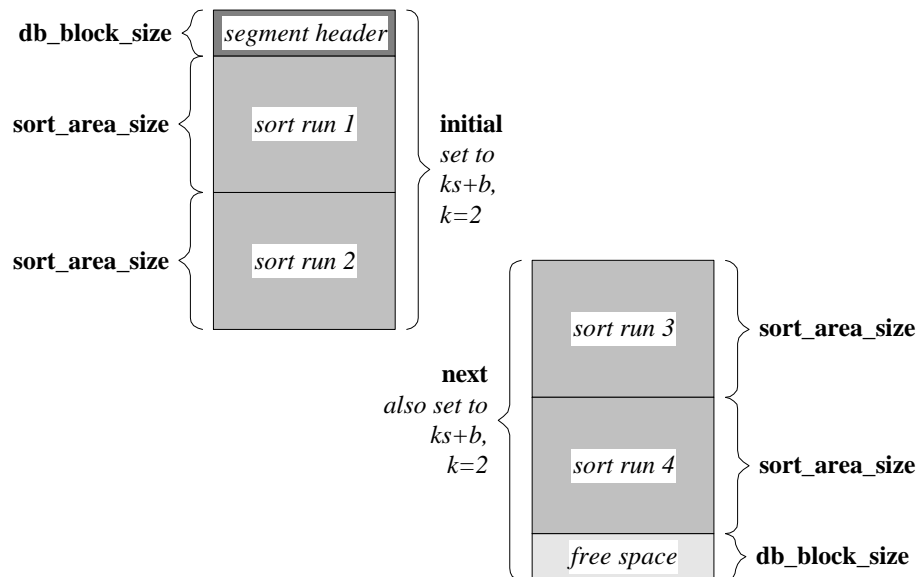
Using **pctincrease 0** ensures that all extents associated with a temporary segment will be the same size, which makes it easier for you to forecast the temporary segment storage requirements of a given process. If you know that some processes will require vastly more temporary space than others, and you don't want to make gigantic the extents used by smaller sorts, then you should consider using more than one tablespace for temporary segments. You can make one tablespace with very large **initial** and **next** default storage parameters and another one with smaller default storage parameters.

Some experienced administrators choose a small **initial**, for which $k = 1$, and a larger value of **next**, for which, say, $k = 8$, under the premise that most of the site's sorts that require disk space are small and can complete with one small temporary segment extent. This plan is designed to save some temporary segment storage.

5.5 Synchronization with the Sort Area

Select a value s for **sort_area_size** such that $s = (E - b) / k$ for some $k = 1, 2, 3, \dots$, where $E = \mathbf{initial} = \mathbf{next}$ is the size of each extent, and b is the value of **db_block_size**.

If you understand how Oracle Server does sorts, you can exploit that knowledge to minimize the number of extent allocations required to perform a sort by matching the size of the sort area to the size of your temporary segment extents. We recommend choosing the sort area size so that one extent will hold a whole number of sort runs with enough room left over for the header block in the temporary segment's first extent, as shown in the following diagram:



Let's consider a specific example. Assume that the **sort_area_size** for a given instance is 256 kilobytes, and that a given database user, whose temporary tablespace is set to **temp**, is executing a sort that will require 1,985 kilobytes of sort area. Assume further that the default

storage parameters on **temp** are (**initial 10k next 10k pctincrease 50**)—the *default* default storage parameters on tablespaces in databases with a 2,048-byte block size. Then completion of the sort would require thirteen extent allocations in **temp**, as shown in the following table:

Extent	Extent Size in 2-Kilobyte Blocks	Segment Size in 2-Kilobyte Blocks	Segment Size in Kilobytes
1	5	5	10
2	5	10	20
3	8	18	36
4	12	30	60
5	18	48	96
6	27	75	150
7	41	116	232
8	62	178	356
9	93	271	542
10	140	411	822
11	210	621	1,242
12	315	936	1,872
13	473	1,409	2,818

Also note that this segment consumes 833 kilobytes more space than actually required to perform the sort. Now consider the extent allocations that would have been required, had the **temp** tablespace default storage parameters been (**initial 512k next 512k pctincrease 0**):

Extent	Extent Size in 2-Kilobyte Blocks	Segment Size in 2-Kilobyte Blocks	Segment Size in Kilobytes
1	256	256	512
2	256	512	1,024
3	256	768	1,536
4	256	1,024	2,048

With these storage parameters, the sort would have required only four extent allocations, and total wasted space would have amounted to only 63 kilobytes.

5.6 Summary of Recommendations

The following list is a summary of the space management techniques we have recommended for temporary segments:

1. For tablespaces that are used for holding temporary segments, choose **initial** and **next** default storage parameter values from the list 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k, 1m, 2m, 3m, and so on.
2. For tablespaces that are used for holding temporary segments, set **initial** and **next** default storage parameters to some number $2^k bm$, where $k = 0, 1, 2, \dots$; b is the value of **db_block_size** for your database; and m is the value of **db_file_multiblock_read_count**.
3. For tablespaces that are used for holding temporary segments, set **initial** and **next** to at least $1.25M/n$, where M is the predicted maximum size of your largest sort to disk, and n is the maximum number of extents a temporary segment can contain.
4. Choose default storage parameters for all tablespaces that will hold temporary segments so that **next** is some integral multiple of **initial**, and use **pctincrease 0**.
5. Select a value s for **sort_area_size** such that $s = (E - b)/k$ for some $k = 1, 2, 3, \dots$, where $E = \mathbf{initial} = \mathbf{next}$ is the size of each extent, and b is the value of **db_block_size**.

6. Estimating Database Space Consumption

This section is devoted to the challenge of estimating how much of the storage allocated to your database is actually required for storing your data.

6.1 The Operating System Perspective

Getting an accurate, detailed answer to this question is a challenge in Oracle7, because the amount of storage *allocated* can vary wildly from the amount of space actually consumed by data. The answer to the question, “How much space does my Oracle database consume on my system?” usually comes from a report such as the UNIX **df** (disk free) report. You might,

for example, find that you have twelve gigabytes of operating system files allocated to Oracle tablespaces.

6.2 The Tablespace Perspective

However, a tablespace can be partly, mostly, or even entirely empty, and you will never know which by looking at an operating system directory listing. The only way to determine how much space inside your tablespaces is actually allocated to Oracle segments (tables, clusters, indexes, etc.) is to run a report like **tf** (tablespace free, shown in sec. 8.9) to determine that a lot of the disk space allocated to database files is currently reserved for future use.

6.3 The Database Block Perspective

However, you cannot tell whether all of the disk space that is allocated to a given segment is actually required by that segment to store data. For example, assume that we were to execute the statement:

```
create table t ...
tablespace users
storage (
  initial      512k
  next        512k
  pctincrease  0
  minextents  1
  maxextents  20
)
```

we would allocate 512 kilobytes of storage to the table named **t**. You could see a report of how many blocks are allocated to **t** by running the **exts** report shown in section 8.1:

```
SQL> @exts cary t
```

But of course, we have just created this table, and in fact none of the space that we have allocated is actually being used to store any rows. How much space does **t** consume? Five hundred twelve kilobytes. But how much space does **t** *require*? Different problem.

To determine how much space a table actually requires is a stiffer challenge than running **tf** or **exts**. To understand whether the blocks allocated to a given segment are used, we need to drill down one level deeper. The following “used block” query will show you the number of database blocks in **t** that have at least one row in them:

```
select
  count(distinct(substr(rowid,15,4) || substr(rowid,1,8)))
from
  table
```

index has required one new extent allocation each month for the past six months, and the extents are one megabyte each, then it's a pretty good bet that the index requires roughly one megabyte per month.

7. Fragmentation

Throughout Oracle's history, Oracle Server administrators have spent a lot of time and effort worrying about Oracle database "fragmentation." Many authors, consultants, and software tool vendors have convincingly marketed the idea that fragmentation hurts Oracle performance. They make a lot of money selling ways to fix the problem.

In truth, fragmentation is not prominent in the well-prepared Oracle Server administrator's work life. Regardless of how difficult and costly it is to fix, Oracle fragmentation is actually easy to prevent. But to prevent fragmentation, you must first understand it. If you are accustomed to using periodic fragmentation removal to "tune" your database, then you will enjoy this chapter. Here, we will explore how you can reduce your server administration costs dramatically by preventing most fragmentation before it occurs. For the times that you actually will have to repair fragmentation problems, we will discuss how to focus your repair techniques on the smallest possible subsets of your database to save time and expense.

The word *fragmentation* itself refers to several very different phenomena in Oracle. Let's explore each one individually.

7.1 Tablespace Free Space Fragmentation

To understand what we mean by *tablespace free space fragmentation*, let us revisit the example we used in section 2.4 to describe extents, segments, and free space, shown again here for convenience:

Tablespace	File#	Block#	Size	Type	Schema.Segment	
users	19	1	1	file hdr		
		2	5	table	cary.a	
		7	5	table	cary.a	
		12	8	table	cary.a	
		20	100	index	cary.a_u1	
			120	5	free	
			125	5	free	
			130	8	free	
			138	375	free	
		20	1	1	file hdr	
		2	512	table	cary.a	
		514	511	free		

Notice that in file 19, there are four adjacent fragments of free space, represented by four rows shown here enclosed in a box. Our intuitions tell us that something wasteful is going on here, because the following abbreviated table would have stored equivalent information about the free space in the tablespace:

Tablespace	File#	Block#	Size	Type	Schema.Segment	
users	19	1	1	file hdr		
		2	5	table	cary.a	
		7	5	table	cary.a	
		12	8	table	cary.a	
		20	100	index	cary.a_u1	
			120	393	free	
		20	1	1	file hdr	
			2	512	table	cary.a
			514	511	free	

The adjacent free space chunks in file 19 are remnants of a segment whose three extents once lived in data block addresses 19.120, 19.125, and 19.130, but which has been dropped with some kind of **drop** statement. It is not possible to determine what kind of segment once lived there, only that there were once at least three extents allocated that have since been de-allocated. The occurrence of free fragments of tablespace is what Oracle literature calls *ta-*

blespace free space fragmentation. We call adjacent free fragments “honeycomb cells,” and we use the term “bubbles” to refer to fragments that are surrounded by in-use extents.

Honeycomb fragments rarely warrant much attention because they affect performance only during **create** statements, and then only very nominally. The nominal performance detriment is due to the additional amount of work that the Oracle7 server process has to perform to “coalesce” adjacent free fragments during segment creation. Excitement about honeycomb fragmentation is rooted in some people’s experience with Oracle Version 6. It used to be that having more than about 60 honeycomb fragments per tablespace in a 2KB-block database incurred a performance problem for people who frequently rebuilt their databases (such as when a training environment is refreshed each week). In Oracle7, this problem no longer exists. In spite of the existence of complex “de-fragmentation tools”¹⁰ and the Oracle7 SMON process (which coalesces honeycomb fragmentation in tablespaces whose **pctincrease** default storage parameter is *non-zero*), “honeycomb” fragmentation is not worth worrying about.¹¹

Bubble fragments also do not harm performance, but isolated “bubbles” of free space can prevent extent allocations from succeeding, in much the same manner as it would be impossible to park a 6-foot-wide car in a parking lot with only 5 feet of space between each pair of cars. There might be 50 total feet of space available in your parking lot distributed in the gaps between eleven cars, but none of it is usable to you if your car won’t fit into the largest open chunk. Fortunately bubble fragmentation is easy to prevent simply by adhering to the rule that segments that will be frequently dropped should not be created in tablespaces containing segments with long life spans [4].

Although neither form of tablespace free space fragmentation is a performance detriment, the appearance of honeycomb and bubble cells is a sure sign that one or more segments have been dropped. Seeing the dead-body outline of a dropped segment in an OFA-compliant production system in a tablespace other than **temp** or **rbs** usually means that a server administrator has dropped and possibly rebuilt an index in the process of optimizing someone’s application. If you adhere to the *OFA Standard* guidelines for configuring your database [4] and use the space management advice given in this paper, you will not have to worry about tablespace free space fragmentation.

¹⁰ I think just about everybody has written one of these. But the truth is that the only real good that mine has done me since the release of Oracle7 is that writing it led me to learn PL/SQL.

¹¹ I would not advise, for example, that you use non-zero **pctincrease** values to impose upon SMON to do de-fragmentation work for you.

7.2 Block Fragmentation

In section 2.6 we discussed the concept of the Oracle7 high-water mark, a marker that allows Oracle7 to forego the reading of table or cluster blocks that have never contained rows. We saw that the high-water mark advances whenever rows are inserted, and that it retreats only when the segment is cleared out by either dropping and recreating it or by truncating it. This “ratchet forward but not back” behavior of the high-water mark leaves opportunity for unnecessary I/O in tables that undergo lots of delete activity.

For example, let’s visit our table called **t** one more time. In **t**’s **create** statement, we initially allocated 512 kilobytes (256 2-kilobyte blocks) of space:

```
create table t ...
  tablespace users
  storage (
    initial      512k
    next        512k
    pctincrease  0
    minextents  1
    maxextents  20
  )
```

In our earlier example (sec. 2.6), we inserted enough rows to fill 2,500 blocks, which required ten total 256-block extent allocations. The high-water mark was set approximately 2,500 blocks deep into the 2,560-block table. A full-table scan of **t** required about 2,500 blocks of I/O.

Now, let’s assume that all the rows are deleted from **t**. This kind of transaction character is common on tables that act as intermediate storage in applications that transfer or summarize data. Since the high-water mark moves only in the direction of insertions, the mark remains at the 2,500-block mark. A full-table scan of this zero-row table would thus require a full 2,500 I/Os. This is why users can sometimes see a simple **select count(*)** that returns **0** take several seconds or even minutes. If we complete the story by inserting an identical copy of the original 2,000 rows, we’ll notice that what once took place almost instantly with 20 blocks of I/O now requires several seconds, while the server scans an additional 2,480 empty blocks. We have just encountered an extreme example of what we call *block fragmentation*. To restore this 2,000-row query to its original timing, we would need to reset the high-water mark.

7.2.1 Repairing Block Fragmentation

In Oracle7, resetting the high-water mark is simple in principal: the Oracle7 **truncate** syntax for tables and clusters is built specifically for this task. However, **truncate** must be used

with care; the statement resets the high-water mark in the delete direction all the way to the end of the segment. It acts functionally as a very high-speed delete that gets rid of all the rows in the table, and which *cannot be rolled back*. So beware. For production tables, be careful that you don't mistakenly delete more than you wish. For interim tables that applications repeatedly fill and empty, you no longer have to drop and re-create your tables (and endure the error-prone process of rebuilding all the appropriate grants and synonyms like you did in Oracle Version 6) just to get optimal full-table scan performance on them.

To truncate a table that is not empty (to which we'll now refer as "rebuilding a table"), simply store the contents of the table in a temporary location, truncate the table, and then restore the contents. If the table does not contain **long** or **long raw** data, then you can use the SQL statements:

```
create table temp as select * from table
truncate table table
insert into table as select * from temp
```

If the table does contain **long** or **long raw** data, then you will need to use **exp** and **imp** or a custom Pro*C program to unload into a file that can be loaded back into the database with SQL*Loader. No matter what the method, when the rows are re-inserted into a freshly **truncated** table, they will be "tightly packed" into as few table blocks as possible, with no row chaining other than that required by extremely long rows.

7.2.2 When to Truncate a Table

Most administrators can't afford to rebuild an entire database—ever. Even to rebuild a table or two requires an availability outage, and the time it takes to execute the process is a linear function of data volume. So, especially with a big database, table rebuilds take a lot of time; hence a server administrator needs to understand how to determine when a table rebuild will actually give enough benefit to offset the cost.

A useful, simple measure of table "badness" that we developed a few years ago is called Ω_1 ("omega-one"). Ω_1 for a given table is defined as follows:

$$\Omega_1 = (h - r) / h = 1 - r/h$$

where h is the number of blocks below the table's high-water mark, and r is the number of blocks in the table that contain at least one row. Computing Ω_1 is straightforward using techniques discussed in section 2.6. Recall that the value of r is the value of **dba_tables.blocks** for the table after an **analyze**, or it can be found by querying **rowid** val-

ues from the table (sec. 2.6.2). And h is the value `dba_segments.blocks` minus the value of `dba_tables.empty_blocks` minus 1 after an `analyze` (sec. 2.6.4).

Intuitively, Ω_1 represents the proportion of table's blocks that are completely empty. A table has an Ω_1 value of 0.0 only if every block in the table up to the high-water mark has at least one row in it, which is good. A table has an Ω_1 value of 1.0 only if every block in the table up to the high-water mark is completely empty, which is as bad as it can get. Non-zero Ω_1 values result from applications that delete rows from tables.

Tables with non-zero Ω_1 values are candidates for truncation. However, don't worry about every table in your application for which $\Omega_1 > 0$. Remember, the purpose of the Oracle table high-water mark is to optimize full-table scans. Non-zero Ω_1 tables that are never searched by full-table scan may be a bit of a space waste, but they are not a performance penalty. If you find a non-zero Ω_1 table that is accessed by a full-table scan, examine the values of h and r before embarking upon a table rebuild.

For example, imagine two tables, **a** and **b**, which are full-scanned heavily as the driving tables in unfiltered SQL joins. Imagine further that the h and r values for the two tables are those shown in the following chart:

	h	r	Ω_1	$h - r$
a	100	51	0.49	49
b	3	1	0.67	2

As you can see, the Ω_1 ratio is worse for **b** than for **a**, but table **a** is really more of a problem, because each full-table scan on **a** wastes 49 blocks of completely non-productive I/O; each full-scan on **b** wastes only two blocks of I/O per full-table scan. You further need to consider the frequency at which these two tables are scanned. For example, if **b** were full-scanned several thousand times in the interior of some nested loop, but **a** were full-scanned only once a day, you have a materially different situation than if neither table had been scanned very frequently.

Glancing at values of Ω_1 helps you identify potential problem tables. It takes knowledge of your application (to know which tables are full-table scanned, and how frequently) and further analysis of $h - r$ (the number of wasted blocks) to know whether a table rebuild is really worth the effort.

7.3 Row Fragmentation

As we have seen, application row deletions can cause Oracle7 block fragmentation—sparsely populated table blocks. Application row updates can also induce another type of fragmentation called *row fragmentation*. Row fragmentation happens when a row is either *migrated* or *chained* [2]. The effect of row fragmentation upon performance is that each time a migrated or chained row is visited in a query, an additional I/O call is required to fulfill the query for each row piece that's not stored in the original block. The performance penalty of row fragmentation can be repaired by rebuilding your database, but it is *not* necessary to rebuild your entire database to get the full benefit of that solution. Oracle7 offers a useful and simple way to detect and correct row fragmentation. You can find complete information in [2], including how to use the **utlchain.sql** utility supplied with Oracle7.

Although the cure for row fragmentation is relatively easy to describe, implementing the cure is much more expensive than proper prevention. You can minimize unnecessary row fragmentation by devoting time to proper selection of **pctfree** values in your application segments. You can find complete information on this topic in [1] and [5].

7.4 Segment Fragmentation

Oracle Server offers an elegant method of space allocation to segments as they are required to grow: Oracle allows a segment to have multiple extents, which the server allocates automatically when they're needed. Presumed performance penalties of having more than one extent per segment has generated considerable debate among Oracle administrators. Many people believe that Oracle Server performance degrades appreciably as the number of extents associated with a segment increases. Many authors speculate that having more than one extent bears a heavy recursive SQL cost. Others cite that costly mechanical disk drive motion is required to scan a table that is not stored in one contiguous hunk. Others cite an alleged devastating impact that multiple extents have on the efficient Oracle Server multi-block read capability.

Probably the most compelling reason for people to believe that a DBMS operates more efficiently if all its segments reside in one extent is that “empirical data” says so:

- People who completely rebuild their databases so that all segments fit into a single extent regularly “see an overall performance gain ranging from five to twenty percent.”
- Queries of multi-extent tables perform “measurably slower” than queries on single-extent tables.

- Segment creates and drops happen “much more quickly” for tables with one extent than for tables with several extents.

The truth is that rebuilding your database so that all segments fit within one extent is not a productive use of your time. Furthermore, attempting to make all your segments have as few extents as possible actually harms your ability to meet your users’ service level requirements, primarily because of the unnecessary downtime imposed by trying to enforce such a policy. Let’s discuss the issues one at a time.

7.4.1 The Mechanics of **exp** “Extent Compression”

The best direct evidence people have for believing that extent fragmentation is bad is that they can routinely get a measurable performance improvement if they use Oracle’s export utility with the **compress=y** option, rebuild their database, and then import their data. The presumption is generally that the act of “compressing” extents resulted in the performance gain. There are two problems here: (1) the performance gain usually deteriorates within three to four months; and (2) the export/rebuild/import process is very expensive.

What people fail to see (only because they don’t try it) is that if they had done the same export/rebuild/import process with **compress=n**, they would have seen an identical temporary performance gain.¹² The performance gains associated with the rebuild cycle come from the following sources:

- Because each table is dropped and rebuilt, each high-water mark is reset to zero. As rows are freshly inserted (the Oracle import utility simply uses the SQL **insert** statement over and over), each table’s high-water mark is moved upward only as much as is necessary to allow the insertions. As a result, when the insertions are complete, each high-water mark in the system is set to its lowest possible value. Consequently, after any import, full-table scans are as efficient as full-table scans can be.
- Because insertions into new data structures pack data as tightly as possible,¹³ when the insertions are complete, all blocks contain exactly as many rows as they are able to contain. Consequently, after any import, every I/O request extracts as much table, cluster, or index data per database block as is physically possible.
- Unless a row is simply too large to fit into one Oracle database block, the only SQL statement that can cause row chaining is **update**. The act of re-inserting every row in the database thus reduces row chaining to its absolute minimum.

¹² The principle at work here is called *multicollinearity*—a result is incorrectly attributed to a cause because two or more potential causes are never experimentally isolated and tested.

¹³ That is, new rows are packed as tightly as possible within the confines of the settings of **pctfree** and **pctused** for the segment.

No wonder the export/rebuild/import cycle results in measurable, if only temporary, performance improvement. So what does the export **compress=y** option really do? It simply replaces the value of **initial** in the **storage** clause of each of your tables, clusters, and indexes with the **blocks** value (converted to bytes) associated with that segment in the **seg\$** dictionary table. It simply causes the **create** statement in the import to try to allocate as much space in one extent as was allocated to the segment when the export occurred. Problems with that include the following:

- The storage parameters that were previously set for your segments are now gone—even ones that you set carefully—replaced with values that don't necessarily suit your wishes.
- The segments that you have intentionally striped across more than one data file will come back unstriped, if they come back at all.¹⁴ If you have striped your data, you have probably done so either for I/O load balancing (you wanted to) or simply because you don't have a single disk drive large enough to hold your largest segments (you had to). Either way, an import of a **compress=y** export will attempt to rebuild each segment back into one extent. If you don't have a data file with a piece of contiguous free space large enough to accommodate that, then your import will simply fail.
- If any segment had been wastefully oversized before the **compress=y** export, then that waste will become institutionalized (i.e., practically permanent because it will be so difficult to detect) after the import. That is, if any segment has ever required an extent allocation—even if the rows that motivated it have since been deleted—the total amount of space ever allocated will now be allocated to the segment's initial extent.
- If you were to constrain yourself to exactly one extent for every segment in your database, then you would have to pre-build all of your segments to have initial extents large enough to handle all of your future growth. This strategy would require you to pre-purchase all of your disk drives for your database, not an especially good plan in a market that gets faster, less expensive disk drives every year.

As you might now expect, a site that experiences a temporary performance improvement by rebuilding its database probably has an opportunity for more permanent performance gain elsewhere. Index and data block sparseness can in fact be a real problem in some delete-intensive applications, and row chaining can be a real problem in an update-intensive application. A solution in either of these areas is usually an applications development challenge, although some problems can be administered effectively by careful resetting of **pctfree** and **pctused** values.

¹⁴ For example, consider a 10-gigabyte table that you have intentionally striped across five 2-gigabyte data files. If you use **compress=y** in an export of this table, the corresponding import will fail when it attempts to create a single 10-gigabyte extent.

However, applications that benefit most from periodic rebuilds are typically those that use inefficient query optimization paths. Sites that use frequent full-table scans routinely gain 5- to 20-percent performance improvements across the board by doing a database rebuild, but SQL statement tuning or addition of optimizing indexes could result in a several thousand percent improvement of selected resource consumers on the system. Once a site gets rid of the full-table scan as its primary data access path, the periodic rebuild becomes neither necessary nor even useful.

The bottom line: (1) optimize your SQL so your applications don't rely on full-table scan access paths; (2) don't do periodic database rebuilds because they're unnecessary, expensive, and risk-inducing; and (3) when you do use the **exp** utility, and it asks you whether to **compress**...just say *no*.

7.4.2 Recursive SQL

After consulting with the most experienced technical specialists in and around Oracle Corporation, it is still not clear how the idea came to be that if you have a large number of recursive calls in your system, you should rebuild your database to reduce the number of extents in your segments. Clearly, there is some overhead in the recursive SQL required to allocate a new extent to a segment on some inserts, and clearly there is some overhead associated with navigating a list of extents during a full-table scan on some queries.

But upon closer inspection, neither issue is really a material argument against multiple extents.

- Yes, inserts into tables and indexes can motivate dynamic extent allocations. But in versions prior to 7.3, there is a limit to the number of extents that can be allocated to a segment for the entire lifetime of an application. Even for an Oracle database with a 8,192-byte block size, a segment can only dynamically extend 504 times, *ever*. Let's make some very pessimistic assumptions for the sake of argument. Let's say that the recursive SQL for dynamic allocation takes as long as 10 seconds (the truth is probably about an order of magnitude less), and let's say that the "set a growth schedule for each segment in your database" rule that's at the heart of this paper is violated so that a segment grows by one extent once a day (about two orders of magnitude more often than it should). Then insertions into the segment would consume a total of ten additional seconds each day, or roughly one hour each year. Considering that a database rebuild of even a very small (e.g., 5-gigabyte) database is a several-hour process, the availability outage for the rebuild exceeds even the most pessimistic performance penalty. A rebuild is not worth the effort.
- And yes, during a full-table scan there is a small overhead associated with calculating the address of the first block in a subsequent extent when the scan exhausts the data in

the current extent of its search. But the effort required to consult a segment's extent map of the segment's header block incurs a zero to very low recursive SQL cost. Especially when we compare the total effort required to do a few dozen (or even a few hundred) new extent address lookups to the magnitude of effort required to full-scan the enormous number of blocks that make up a multi-extent table, we find that the overhead rounds to zero.

By far, the SQL operations that generate the most intense recursive SQL demands are DDL operations like **create** and **drop**.

7.4.3 Create and Drop

The number of extents in a segment does make a measurable difference in performance of DDL that creates or drops a segment. DDL is really just a macro language that makes it easy for us to, for example, drop a table without having to remember how to do all sorts of insert, update, and delete magic on database dictionary tables like **obj\$, tab\$, ind\$, icol\$, cdef\$, ccol\$, seg\$, uet\$, fet\$**, and so on. The more extents that are associated with a segment, the more dictionary rows that will have to be manipulated during a DDL operation. Consequently, DDL on multi-extent segments will perform less quickly than DDL on single-extent segments.

The question then becomes: How appropriate is it for an application to create or drop a table? Unfortunately, in Oracle Version 6 there were several motives to drop and rebuild a table within an application rather than simply delete from it and begin inserting again. Oracle7 obviates many of those motives by introducing the **truncate** command. Well-designed Oracle7 applications rarely, if ever, execute DDL during normal application operation. DDL performance, therefore, should not influence your decision about whether a large table, cluster, index, or rollback segment in your applications should be allowed to consume more than one extent.

7.4.4 Drive Head Motion

Another popular argument against having multiple extents is that a disk drive head execut-

Unfortunately for the applicability of the experimental results in OLTP environments, almost all such tests are performed in a carefully controlled environment on a single-user computing system. To understand the profound influence of this assumption on the argument, let's review the mechanics of a full-table scan (we'll hit the high spots).

1. A server process will read a batch of blocks from a database file, and those blocks are pinned into the database buffer cache of the SGA.
2. The server process sequentially reads information from the blocks in memory, processes that information, returns rows, and un-pins the blocks in the SGA.

Step 2 requires non-zero time, and especially if **db_file_multiblock_read_count** is large, finishing step 2 gives plenty of time for a competing process on the system to have queued an I/O request of the disk drive that our experimenters assume to be patiently awaiting our next I/O request. Fact is, our disk drive head is just as likely to be close to the first block of a new extent as it is to be close to the next block in our current extent. Experiments on heavily-loaded, operational OLTP computing systems show there to be no measurable difference between full-table scans on large tables with one extent and full-table scans on large tables with many extents.

Experiments on operational computing systems that are dedicated to a single large batch process performing full-table (sequential) scan queries do show some measurable difference in performance between tests in which all tables fit into one large extent and tests in which tables contain more than one extent. However, the detrimental effects of having more than one extent per segment can be practically eliminated by:

- ensuring that your extent sizes are appropriately synchronized with the value of **db_file_multiblock_read_count** (sec. 3.2);
- ensuring that your extent sizes are not pathologically small (sec. 3.3);
- ensuring that you don't have pathologically many extents (sec. 3.5); and
- eliminating unnecessary **create** and **drop** operations from the application design (by using **truncate** instead).

7.4.5 Direct Access Paths

Most good applications rely exclusively on non-sequential access paths; full-table scans are an inefficient data access path that are often avoidable by using indexes. What is the impact of extent fragmentation on indexed reads? Fortunately the answer is that there is absolutely no impact. Multiple extents in tables and indexes do not penalize direct access (index- or hash-based) read performance at all.

Index traversal is executed entirely by Oracle *data block address* (DBA), beginning with an index root node access, and traversing a B*-tree by direct I/O of exactly the needed block. It does not matter whether a two index blocks are in the same extent or not, because each is read with a separate read call. Direct access data blocks are also fetched by DBA. Two or more data blocks fetched in the same query thus will motivate separate read calls as well. It does not matter whether two such blocks are in the same extent or not, the read times are statistically identical.

7.4.6 Real Benefits of Multiple Extents

There are actually several very good reasons to use more than one extent in an Oracle database segment. To summarize:

- Using more than one extent in a table that is never full-scanned bears absolutely no impact on the performance of queries on that table.
- Using more than one extent in an index bears absolutely no impact on the performance of searches performed using that index.
- Using more than one extent in a table, cluster, or temporary segment does not materially impact the performance of full-scans on an operational multi-user system.
- Using more than one extent in a table, cluster, or temporary segment does not materially impact the performance of full-scans on a dedicated single-user batch processing system if the extents are well-sized and if the application is written to avoid expensive DDL operations.
- If you match your extent sizes appropriately to your multi-block read batch size, you further minimize the alleged performance cost of having many extents in a segment.
- You should *prefer* many extents to few extents for rollback segments, because using multiple extents for rollback segments can help reduce recursive SQL calls to do dynamic extent allocations on the segments.
- The real constraint on the number of extent allocations you want for a segment in versions of Oracle prior to 7.3 is the segment's effective **maxextents** value, which permanently limits the growth of a segment, and which has an upper bound that is a function of your database block size.
- It isn't possible to put very large segments into single extents because of file size and file system size limitations.
- You should want to put your largest segments into many extents, because it gives you the opportunity to stripe parts of those segments across different disk drives.

- You should want segments to allocate new extents over time, to allow you to take advantage of the market's faster, less expensive disks.

8. Selected SQL Source Code

The SQL source code listed in the following sections performs some of the tasks described in this paper. Enjoy.

8.1 exts.sql — Extents Summary

```
rem exts.sql - print segment storage summary
rem
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1993 by Oracle Corporation
rem @(#)exts.sql 1.2 (93/11/07)

def ownr      = &&1
def segt      = &&2

col ownr form      a8 head 'Owner'          just c
col name form      a28 head 'Segment Name'  just c
col type form      a8 head 'Type'          just c trunc
col hfil form      9,990 head 'Header|File' just c
col hblk form      99,990 head 'Header|Block' just c
col exts form      9,990 head 'Extents'     just c
col blks form      999,990 head 'Blocks'    just c

break -
  on ownr skip 1

select
  owner      ownr,
  segment_name name,
  segment_type type,
  header_file hfil,
  header_block hblk,
  extents    exts,
  blocks     blks
from
  dba_segments
where
  owner like upper('&ownr')
  and
  segment_name like upper('&segt')
```

```

/
undef ownr
undef segt

```

8.2 se.sql — Segment Extent History

```

rem se.sql - print segment extension history
rem
rem Dominic Delmolino, Oracle Corporation
rem Copyright (c) 1994 by Oracle Corporation
rem @(#)se.sql 1.1 (94/01/04)

def ownr          = &&1
def segt          = &&2

col ownr form          a8 head 'Owner'          just c trunc
col name form         a28 head 'Segment Name'    just c trunc
col type form         a5 head 'Type'            just c trunc
col stmp form         a6 head 'Date'            just c
col exid form         990 head 'Ext'            just c
col byts form 999,999,990 head 'Bytes'          just c
col blks form 999,990 head 'Blocks'            just c

break -
  on ownr skip 1 -
  on name -
  on type -
  on stmp

select
  owner                ownr,
  segment_name         name,
  segment_type         type,
  to_char(extend_date,'YYMMDD') stmp,
  extent_id           exid,
  bytes               byts,
  blocks              blks
from
  aps_segment_extends
where
  owner like upper('&ownr')
  and
  segment_name like upper('&segt')
order by
  owner,
  segment_name,
  segment_type,
  extent_id

```

/

```
undef ownr
undef segt
```

8.3 se.tbl — Create Objects for se Utilities

```
rem se.tbl - create required objects for se utilities
rem
rem Dominic Delmolino, Oracle Corporation
rem Copyright (c) 1994 by Oracle Corporation
rem @(#)se.tbl.sql      1.1 (93/11/06)
rem
```

```
insert into aps_segment_extends (
  owner,
  segment_name,
  segment_type,
  extend_date,
  extent_id,
  bytes,
  blocks
)
select
  da.owner,
  da.segment_name,
  da.segment_type,
  sysdate,
  da.extent_id,
  da.bytes,
  da.blocks
from
  dba_extents da
where
  da.owner like upper('&ownr')
  and
  da.segment_name like upper('&segt')
  and
  da.extent_id > (
    select
      nvl(max(ase.extent_id),-1)
    from
      aps_segment_extends ase
    where
      ase.owner          = da.owner
      and
      ase.segment_name = da.segment_name
      and
      ase.segment_type = da.segment_type
  )
/

select
  'Segment history refreshed for ' ||
  upper('&ownr') || '.' || upper('&segt') line
from
  dual
/

undef ownr
undef segt
```

8.5 sedel.sql — Delete Segment Extent History

```

rem sedel.sql - delete segment extension history
rem
rem Dominic Delmolino, Oracle Corporation
rem Copyright (c) 1994 by Oracle Corporation
rem @(#)sedel.sql 1.1 (94/01/04)

def ownr          = &&1
def segt          = &&2

col line form a75          head 'Status'

delete from
  aps_segment_extends
where
  owner like upper('&ownr')
and
  segment_name like upper('&segt')
/

select
  'Segment history deleted for ' ||
  upper('&ownr') || '.' || upper('&segt') line
from
  dual
/

undef ownr
undef segt

```

8.6 spseg.sql — Storage Parameters for Segments

```

rem spseg.sql - report segment extent storage parameters
rem
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1993 by Oracle Corporation
rem @(#)spseg.sql 1.2 (93/11/07)

def ownr          = &&1
def segt          = &&2

col ownr form      a8 head 'Owner'          just c
col name form      a28 head 'Segment Name' just c
col type form      a8 head 'Type'          just c trunc
col init form 999,990 head 'Initial|(KB)' just r
col next form 999,990 head 'Next|(KB)'     just r
col pcti form      990 head 'Pct|incr'     just r

```

```
col minx form          990 head 'Min|exts'      just r
col maxx form         990 head 'Max|exts'      just r
```

```
break -
  on ownr skip 1
```

```
select
  owner                ownr,
  table_name           name,
  'TABLE'              type,
  initial_extent/1024  init,
  next_extent/1024     next,
  pct_increase         pcti,
  min_extents          minx,
  max_extents          maxx
from
  dba_tables
where
  cluster_name is null
  and
  owner like upper('&ownr')
  and
  table_name like upper('&segt')
union
select
  owner                ownr,
  index_name           name,
  'INDEX'              type,
  initial_extent/1024  init,
  next_extent/1024     next,
  pct_increase         pcti,
  min_extents          minx,
  max_extents          maxx
from
  dba_indexes
where
  owner like upper('&ownr')
  and
  index_name like upper('&segt')
union
select
  owner                ownr,
  cluster_name         name,
  'CLUSTER'           type,
  initial_extent/1024  init,
  next_extent/1024     next,
  pct_increase         pcti,
  min_extents          minx,
  max_extents          maxx
from
```

```

    dba_clusters
where
    owner like upper('&ownr')
    and
    cluster_name like upper('&segt')
union
select
    owner                ownr,
    segment_name         name,
    'ROLLBACK'          type,
    initial_extent/1024  init,
    next_extent/1024    next,
    pct_increase         pcti,
    min_extents          minx,
    max_extents          maxx
from
    dba_rollback_segs
where
    owner like upper('&ownr')
    and
    segment_name like upper('&segt')
order by
    1
/

undef ownr
undef segt

```

8.7 spts.sql — Storage Parameters for Tablespaces

```

rem spts.sql - report on tablespace storage parameters
rem
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1991,1993 by Oracle Corporation
rem @(#)spts.sql 1.2 (93/11/07)

col stat form      a7 head 'Status'      just c
col name form      a15 head 'Tablespace'  just c
col init form 999,990 head 'Initial|(KB)' just c
col next form 999,990 head 'Next|(KB)'    just c
col pcti form      990 head 'Pct|incr'    just c
col minx form      990 head 'Min|exts'    just c
col maxe form      990 head 'Max|exts'    just c

break -
    on stat skip 1

select
    status                stat,

```

```

    tablespace_name      name,
    initial_extent/1024  init,
    next_extent/1024    next,
    pct_increase         pcti,
    min_extents         minx,
    max_extents         maxe
from
  dba_tablespaces
where
  status!='INVALID'
order by
  1,2
/

```

8.8 tmap.sql — Tablespace Map by Block

```

rem tmap.sql - print map of database blocks
rem
rem Grant Franjione, Oracle Corporation
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1991,1995 by Oracle Corporation
rem @(#)tmap.sql 1.3 (95/06/02)

def ts          = &&1

col tablespace form          a15 head 'Tablespace' just c trunc
col file_id    form          990 head 'File'          just c
col block_id   form 9,999,990 head 'Block Id'        just c
col blocks     form          999,990 head 'Size'       just c
col segment    form          a38 head 'Segment'      just c trunc

break -
  on tablespace skip page -
  on file_id skip 1

select
  tablespace_name          tablespace,
  file_id,
  1                          block_id,
  1                          blocks,
  '<file hdr>'              segment
from
  dba_extents
where
  tablespace_name = upper('&ts')
union
select
  tablespace_name          tablespace,
  file_id,

```

```

1                block_id,
1                blocks,
'<file_hdr>'     segment
from
  dba_free_space
where
  tablespace_name = upper('&ts')
union
select
  tablespace_name      tablespace,
  file_id,
  block_id,
  blocks,
  owner||'.'||segment_name  segment
from
  dba_extents
where
  tablespace_name = upper('&ts')
union
select
  tablespace_name  tablespace,
  file_id,
  block_id,
  blocks,
  '<free>'
from
  dba_free_space
where
  tablespace_name = upper('&ts')
order by
  1,2,3
/

undef ts

```

8.9 tf.sql — Tablespace Free Space

```

rem tf.sql - report free tablespace
rem
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1990,1992 by Oracle Corporation
rem @(#)tf.sql      1.2 (93/11/07)

col tsname form          a16 head 'Tablespace'      just c
col nfrags form          999,990 head 'Free|Frag'    just c
col mxfrag form 999,999,990 head 'Largest|Frag (KB)' just c
col totsiz form 999,999,990 head 'Total|(KB)'      just c
col avasiz form 999,999,990 head 'Available|(KB)'  just c
col pctusd form          990 head 'Pct|Used'       just c

```

```

select
  total.tablespace_name          tsname,
  count(free.bytes)              nfrags,
  nvl(max(free.bytes)/1024,0)    mxfrag,
  total.bytes/1024               totsiz,
  nvl(sum(free.bytes)/1024,0)    avasiz,
  (1-nvl(sum(free.bytes),0)/total.bytes)*100 pctusd
from
  aps_data_files total,
  dba_free_space free
where
  total.tablespace_name = free.tablespace_name(+)
group by
  total.tablespace_name,
  total.bytes
/

```

8.10 tf.tbl — Create Objects for **tf** Utility

```

rem tf.tbl - create required objects for tf utility
rem
rem Cary Millsap, Oracle Corporation
rem Copyright (c) 1990,1992 by Oracle Corporation
rem @(#)tf.sql      1.2 (93/11/07)

rem * This program should be run from the SYS schema.

rem * tf.sql
rem *
drop view aps_data_files;
create view aps_data_files as
  select
    tablespace_name,
    sum(bytes)      bytes
  from
    dba_data_files
  group by
    tablespace_name
/
grant select on aps_data_files to public;
drop public synonym aps_data_files;
create public synonym aps_data_files for aps_data_files;

```

8.11 users.sql — User Privileges

```

rem users.sql - report granted priveleges to users

```

```

rem
rem Craig Shallahamer, Oracle Corporation
rem Copyright (c) 1990,1993 by Oracle Corporation
rem @(##)users.sql 1.3 (93/11/07)

col username form a12 head 'Username'           just c
col role      form a21 head 'Role (admin,grant)' just c
col dts       form a12 head 'Default|Tablespace' just c
col tts       form a12 head 'Temporary|Tablespace' just c
col prof      form a18 head 'Profile'           just c

break -
  on username skip 1 -
  on role -
  on dts -
  on tts

select
  username,
  default_tablespace dts,
  temporary_tablespace tts,
  profile prof,
  granted_role || '-' ||
  decode(admin_option,'YES','A',' ') ||
  decode(granted_role,'YES','G',' ') role
from
  dba_users,
  dba_role_privs
where
  dba_users.username = dba_role_privs.grantee and
  username not in ('PUBLIC')
order by
  1,2,3,4
/

```

9. References

- [1] *Oracle7 Server Concepts Manual*. Oracle Part No. 6693-70-1292 December 1992.
- [2] *Oracle7 Server Administrator's Guide*. Oracle Part No. 6694-70-1292 December 1992.
- [3] *Oracle7 Server Utilities User's Guide*. Oracle Part No. 3602-70-1292 December 1992.
- [4] MILLSAP, CARY V. *The OFA Standard, Oracle7 for Open Systems*. Oracle Part No. A19308-1 May 1994.

- [5] SHALLAHAMER, CRAIG A. "Avoiding a Database Reorganization" in *Oracle Magazine*, Vol. VIII No. 4, pp. 63–70, Fall 1994.



Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, California 94065
415.506.7000
800.442.8649
Fax 415.506.7200

International inquiries:
44.932.872.020
Telex 851.927444 (ORACLE G)
Fax 44.932.874.625

Copyright © Oracle Corporation 1994, 1995
All Rights Reserved
Printed in the U.S.A.

A00000-0 (no Oracle part number has been assigned)