

HOW TO STOP DEFRAGMENTING AND START LIVING: THE DEFINITIVE WORD ON FRAGMENTATION

Bhaskar Himatsingka, Oracle Corporation
Juan Loaiza, Oracle Corporation

ABSTRACT

Fragmentation is an issue of great concern to database administrators. Oracle has a multitude of commands, options, views, and statistics that address various aspects of fragmentation, but it is not obvious how these capabilities should be used on a day-to-day basis. There is also a great deal of partial information, obsolete information, and misinformation that has been published on this subject that can confuse and frustrate administrators. Most of this material was written before Oracle7 release 7.3 became available and commonly used. New functionality in Oracle7 release 7.3 reduces fragmentation issues and allows different techniques to be used that are both simpler and more effective. This paper describes Oracle's capabilities related to fragmentation and gives concrete and specific advice on how these capabilities should be applied.

This paper is targeted at an audience of experienced database administrators. It is based on Oracle7 release 7.3 and also covers new features introduced in Oracle8 version 8.0.

1. Introduction

Oracle manages disk space as a collection of user supplied files referred to as datafiles.

Schema objects which occupy disk space e.g. tables, indexes, clusters, etc. consist of a collection of blocks. For performance and manageability reasons Oracle has the notion of segments which are a collection of blocks private to a schema object. Multiple segments are allocated for a schema object if it is horizontally partitioned (using range, hash, etc.) or vertically partitioned i.e. has large objects, collections, or an overflow segment.

Oracle also allows for the grouping of the datafiles into logical entities called tablespaces. Tablespaces also provide for the compartmentalization of disk space usage amongst various database objects, by ensuring that all blocks belonging to a segment are only allocated from files within one tablespace, i.e. a segment is local to a tablespace.

Thus a database can be divided into multiple tablespaces based on the requirements of the different data sets placed in the database. Schema objects themselves can be divided or more precisely, partitioned again based on performance and manageability requirements. Individual partitions or more precisely segments can be placed across the various tablespaces as there is no requirement that all segments for a schema object reside in the same tablespace. In fact for managing very large tables, Oracle recommends placing the segments in multiple tablespaces.

In the following sections, we discuss how Oracle does space management for these entities. In section 2, we discuss some simple management techniques that can be used to avoid fragmentation at the tablespace level. Section 3 discusses fragmentation issues for heap tables and again attempts to provide simple steps to monitor heap fragmentation. Section 4 goes on to discuss similar issues for B+Tree (indexes). Section 5 talks about using partitioning as a means of eliminating fragmentation. Lastly we conclude this paper with a recap of the whole discussion.

2. Tablespace Management

As discussed in the previous section, a tablespace is a collection of datafiles which represent the disk space (or blocks) which Oracle can consume. Schema objects occupy space in a tablespace using segments which are a collection of

blocks belonging to that tablespace. There can be multiple segments in a tablespace and these segments can belong to the same or multiple schema objects.

So what causes fragmentation? The number of blocks belonging to a segment can be quite large. To manage access to these blocks more efficiently Oracle further groups them together as extents. An extent refers to a collection of contiguous blocks in a file. Hence, a segment ends up being a collection of extents. The advantages of allocating space as extents instead of individual blocks are:

- *Efficient Allocation* - An extent is a very efficient way to represent a large number of blocks since it just lists the start block and length instead of listing every block. The time it takes to allocate and deallocate space in Oracle is proportional to the number of extents that are manipulated, not the number of blocks. Therefore large segments consisting of thousands or millions of blocks can be allocated and deallocated quickly because the blocks can be represented as a small number of extents.
- *Efficient Scans* - The fact that blocks in an extent are contiguous in the underlying file allows Oracle to issue very efficient multi-block read operations when a segment is scanned.

The disadvantage of using extents is that the space in a tablespace can become fragmented.

- *Extent Fragmentation* - Extent fragmentation refers to fragmentation that occurs at the tablespace level between extents. A tablespace is fragmented when enough space exists in the tablespace to satisfy a request but the space is spread across several free extents and therefore cannot be used to satisfy the request. For example, a tablespace might have 100 free extents of 10 megabytes each, but if none of these extents are contiguous, a 40 megabyte request cannot be satisfied even though there is a gigabyte of free space in the tablespace. When this happens, we say that there is extent fragmentation. At this point either the user can reduce the desired extent size or move data (extents) to make more contiguous free space available. Neither option is particularly attractive. The best solution is to avoid fragmentation in the first place.

2.1 Eliminating Extent Fragmentation Using SAFE

Oracle provides administrators extreme flexibility in sizing the extents that are allocated to segments. This full specification flexibility is almost never needed and in fact, should be avoided in most cases. By following a simple set of administrative rules, fragmentation at the extent level can be entirely eliminated. We will use the term SAFE to refer to these rules. SAFE stands for Simple Algorithm for Fragmentation Elimination. SAFE consists of a set of rules, many of which have been internalized and implemented as part of the locally managed tablespace feature being introduced in Oracle8i. Oracle8i users should refer to the presentation on that subject for an updated version of SAFE.

2.1.1 Use Uniform Extent Sizes

All the segments in a tablespace should have exactly the same extent size. This insures that any free extent in a tablespace can always be used for any segment in the tablespace. Thus, there will never be an extent so small that it cannot be chosen. This implies that the NEXT value of all storage clauses should be set to be the same as the value of INITIAL, and the PCTINCREASE value should always be set to zero.

To further insure that extent sizes are uniform, Oracle8 added the 'MINIMUM EXTENT' clause to Create Tablespace. Specifying a value for MINIMUM EXTENT insures that all allocated extents in the tablespace are a multiple of that value. If the extent being allocated would normally be smaller, then it will be automatically rounded up to be a multiple of the MINIMUM EXTENT value. Also, if an extent is being trimmed, for example as a result of a parallel direct load, then the remaining size and the trimmed size will be set to multiples of MINIMUM EXTENT. Always specify a value for MINIMUM EXTENT that is equal to the value of INITIAL to insure uniform extent sizes.

2.1.2 Specify Storage Clauses Only and Always at the Tablespace Level

In order to insure that all the segments within a tablespace have exactly the same extent size, you should never explicitly specify a storage clause at the segment level. Instead, let the storage values for all segments be inherited from the storage clause of the tablespace. Always explicitly set the storage clause at the tablespace level. If it is required that the segment has a guaranteed amount of space when it is created then specify MINEXTENTS in the storage clause for the segment creation. This will allocate multiple extents all of the same size (the one for the tablespace) to the segment.

Caution - When a segment is exported, the storage clause for the segment is populated in full by export. Thus exporting a segment from one tablespace to another when they have different extent sizes breaks this rule. There are no obvious capabilities in Oracle to deal with this problem. We recommend not using the compress=y feature of export so that this issue only arises for tablespaces with different extent sizes. When the segment is moved across extent sizes, the only way to be SAFE is to precreate the segment and then do the IMPORT with ignore=y.

2.1.3 The Extents Size for all Data Tablespaces Should be one of 160K, 5120K, or 160M

These three extent sizes are all that you will ever need for tables and indexes. Using more than these three extent sizes will not significantly improve performance or space utilization but it will complicate administration and potentially induce fragmentation.

The smallest extent size, 160K, is large enough to make scanning efficient. It is small enough that disk wastage is insignificant for even small tables. One might think that 160K is a lot of disk space to waste on segments that only have a very small amount of data. However, in practice, the labor required to manage smaller extents is not worth the small savings in disk cost. For example, if a database has ten thousand tiny tables with an extent size of 160K, then a maximum of 1.6G of disk will be wasted. The cost of this much disk will be swamped by the labor cost of dealing with fragmentation and inefficient scans for the lifetime of the database. Also, if the segment only uses a few blocks, the remaining blocks will be kept above the 'high water mark' of the segment and therefore will not hurt performance during scans. Unless your database has many tens of thousands of very small segments, don't use an extent size smaller than 160K.

The medium extent size, 5120K, is large enough to handle all but the very largest segments of large databases. The largest extent size, 160M, is large enough to handle segments of up to hundreds of gigabytes in size efficiently. As we will discuss later, segments larger than this should be avoided when possible. This extent size should be used for only the largest tables that contain many Gigabytes of data. As a rule, this extent size will mostly be used for very large segments that cannot be partitioned either because partitioning is not available, or because the segment type does not allow partitioning.

When you create a new segment, you need to have a very rough estimate of the ultimate size of the segment in order to assign it to a tablespace with the appropriate extent size. Follow these three simple rules when assigning a segment to a tablespace.

- 1) Segments smaller than 160M should be placed in 160K extent tablespaces.
- 2) Segments between 160M and 5120M should be placed in 5120K extent tablespaces.
- 3) Segments larger than 5120M should be placed in 160M extent tablespaces.

These extent sizes are chosen to be a multiple of five blocks since Oracle7 will round all extent sizes to a multiple of five blocks. In Oracle8 extents won't be rounded to a multiple of five blocks if they are a multiple of the minimum extent size for the tablespace. For Oracle8 databases, choosing the following extent sizes is a little simpler.

- 1) Segments smaller than 128M should be placed in 128K extent tablespaces.
- 2) Segments between 128M and 4G should be placed in 4M extent tablespaces.
- 3) Segments larger than 4G should be placed in 128M extent tablespaces

For the remainder of this paper we will assume that the Oracle7 extent sizes are used since they can be used in both releases.

2.1.4 Monitor and Potentially Relocate Segments Having More Than 1024 Extents

Oracle supports an unlimited number of extents in a segment. The performance for DML operations is largely independent of the number of extents in the segment. However, certain DDL operations such as dropping and truncating of segments are sensitive to the number of extents. Performance measures for these operations have shown that a few thousand extents can be supported by Oracle without a significant impact on performance. A reasonable maximum has been determined to be 4096.

The goal of our recommended algorithm is to keep the number of extents below 1024 which is well within the range that Oracle can efficiently handle. When a segment reaches 1024 extents it is a candidate to be moved to the next larger extent size tablespace. The segment does not necessarily have to be moved immediately or at all. The segment may be near its peak steady state size, in which case even if it has a few thousand extents, it should be left where it is. It is only the segments which are growing that have to be targeted and potentially moved to tablespaces with larger extents.

The three recommended extent sizes 160K, 5120K, and 160M differ from each other by a factor of 32. Therefore, moving a segment with 1024 extents to the next larger extent size tablespace will cause the number of extents in the segment to be reduced by a factor of 32 to 32 extents. 32 extents is large enough that the loss of space due to unused blocks in the segment will be limited to 3 percent of the total space in the segment and the expected loss will be 1.5 percent. This is small enough to ignore in practice.

It is a common misconception that there is an advantage to having all the space for a segment be in one or a small number of extents. In fact, the real advantage is gained by using large extents and not by having a small number of extents. By using large extents, scan operations can issue large contiguous disk reads. Extent sizes of 160K already capture most of this speedup. 5120K extent sizes capture all of this speedup. Most segments are only ever accessed through indexes, so the table scan time is totally irrelevant anyway.

Remember that segments using 120K extent sizes are likely to be small which means that, if they are frequently accessed, they will be kept in the buffer cache and thus will not be scanned from disk. It is occasionally advantageous to trade off unused blocks for better scan times by allocating segments that would normally use 160K extents in 5120K extent tablespaces instead. However, this should be done only rarely since it seldom improves performance significantly.

Another class of segments that might be placed in a tablespace with larger extents is the class of 'volatile' segments. Volatile segments are segments that are frequently dropped or truncated. Tables that hold temporary results are an example of volatile segments. Since volatile segments frequently allocate and free extents, their performance can be slightly improved by placing them in a tablespace having larger extents and thus reducing the number of extents.

2.1.5 The Maximum Single Segment Size Should be Somewhere between 4G and 128G

Segments that are very large become a problem for performance, manageability, and recovery. The limit that should be placed on the maximum size of a segment varies by installation depending on the hardware throughput and database availability requirements but it is somewhere in the range of 4G to 128G. Segments larger than this should be partitioned into multiple segments when possible. 160M extent tablespaces can contain segments up to 160G and still have less than 1024 extents per segment.

2.1.6 Very Large Tables and Indexes should be Placed in a Private Tablespace

Tables or indexes that are very large become a problem for performance, manageability, and recovery. The definition of a table or index that is very large varies by installation depending on hardware throughput and database availability

requirements but is somewhere in the range of 4G to 128G. Tables or indexes that are very large should be placed in a set of private tablespaces that don't contain any other data. This applies even if the table or index is partitioned into segments that are smaller than 4G. Doing this will make management easier and enable the use of tablespace point in time recovery under certain conditions in the event of a serious problem.

2.1.7 Temp Segments Should be Restricted to TEMP Tablespaces

A temp segment is a segment that is created automatically by Oracle to hold a partial result. Temp segments are created by sort operations that spill to disk, by hash-join operations that spill to disk, and by some types of referential integrity enforcement operations that spill to disk. The TEMPORARY TABLESPACE clause of the CREATE USER command controls the tablespace that will be used when a temp segment is created. This tablespace should always be set to a tablespace that contains no permanent data and that is explicitly declared to be a temporary tablespace using the TEMPORARY clause of CREATE TABLESPACE. Tablespaces declared to be temporary use special space management algorithms that are more efficient for management of temp segments and that avoid fragmentation.

Caution - Oracle8 does not provide a means to define a tablespace to be the default temporary tablespace for the database. Thus, to be SAFE it is imperative that the TEMPORARY TABLESPACE clause of CREATE USER always be specified whenever a new user is created. The temporary tablespace for users SYS and SYSTEM should also be altered to the temporary tablespace of the database.

2.1.8 Place Rollback Segments in Tablespaces dedicated to Rollback Segments

Rollback segments should be placed in tablespaces containing only other rollback segments. This allows rollback segments to grow and shrink without fragmenting the extents in the other tablespaces. Also, rollback segments are critical to the functioning of Oracle and therefore should be located on the most robust disk storage available. Furthermore, if tablespace point in time recovery is ever needed, the rollback segment tablespaces will need to be restored along with the data tablespaces to be recovered. For all these reasons it is highly recommended that separate tablespaces be created to hold the databases rollback segments. We will call these tablespaces UNDO tablespaces.

2.1.9 TEMP and UNDO Tablespaces should contain between 1024 and 4096 extents

TEMP tablespaces and UNDO tablespaces have unique space allocation requirements. The number of extents in these tablespaces is more important than their size. The number of extents in these tablespaces should be set based on the number of concurrent users in the database. This is because idle systems don't use any space in these tablespaces, but busy systems might require space for each concurrent user. Sometimes a single user will consume almost all the space in an UNDO or TEMP tablespace by growing a single segment to the full size of the tablespace. For efficiency, we want to keep the number of extents in such a segment to no more than 4096. Other times many users will consume a small amount of space each. In this case we want to use relatively small extents so that many independent extent allocations can be made. We also wish to leave room for growth when initially sizing these tablespaces. Therefore, we recommend that the number of extents in UNDO and TEMP tablespace should initially be set to 1024. To do this, set the extent size to the total tablespace size divided by 1024. Allowing 1024 extents to be allocated within a tablespace handles both the single large user and many small users cases efficiently.

Each concurrently active user of a TEMP tablespace will need to allocate at least one private extent. If there is a mixed work load, e.g. there are some users running smaller sorts, while others run large jobs, then we would like to prevent the users running small sorts from consuming most of the available space. Therefore, the number of active users concurrently using a TEMP tablespace should be no more than one quarter to one half the number of extents in the tablespace. This insures that most of the space is available for larger jobs even if each small job consumes one extent. You should either

increase the number of extents or create multiple tablespaces to enforce this rule. As discussed earlier, the number of extents can be comfortably increased to 4096 beyond which you should add more tablespaces.

Similarly, the number of rollback segments per tablespace should be restricted. Every rollback segment should have its `OPTIMAL` parameter set so that it contains at least four extents. This is so that a single transaction can use at least three fourths of the space in the rollback segment without having to extend it. Therefore, the number of rollback segments per undo tablespace should be limited to one eighth to one sixteenth the number of extents in the tablespace. This allows each rollback segment to have four extents while still leaving at least half the space in the undo tablespace for large transactions.

When you add more disk space to `UNDO` and `TEMP` tablespaces you should check to see if the number of extents in the tablespace will exceed 4096. If so, then you can either add a new tablespace or grow the extents of the existing tablespace. Growing the extents of an existing tablespace can be accomplished by changing the storage parameters for that tablespace as follows:

```
ALTER TABLESPACE tbspc_name
DEFAULT STORAGE (INITIAL new_extent_size NEXT new_extent_size
                 PCTINCREASE 0, MAXEXTENTS 4096)
MINIMUM EXTENT new_extent_size;
```

If the tablespace is an `UNDO` tablespace then you will need to drop all the existing rollback segments and recreate them so that the new extent size will be used.

2.1.10 Never Place User Data in the System Tablespace

The system tablespace contains database meta-data. Oracle recommends that no user schema objects be created in this tablespace. The segments in the system tablespace will never be dropped or truncated. Therefore extent fragmentation cannot occur in the system tablespace as long as users don't add user schema objects to this tablespace. The extent sizes for segments in the system tablespace are managed by Oracle to insure that space is not over allocated to tiny segments.

Caution - Oracle8 does not follow the `SAFE` algorithm for the `SYSTEM` tablespace. Hence, it is important that user segments not be created in this tablespace as that will surely lead to fragmentation amongst other problems. System segments which tend to grow e.g. segments which hold data for pl/sql packages etc. follow a somewhat `SAFE` scheme and hence should not cause fragmentation in most installations.

2.1.11 Size Files to be a multiple of the tablespace extent size plus 1 block

The only possible extent fragmentation in a tablespace with uniform extent sizes is fragmentation at the end of every file because the extent size does not match the size of the file. Oracle uses the first block of every file to maintain internal bookkeeping information. Therefore, when allocating files to a tablespace, make sure to set the size of the file to be a multiple of the extent size for the tablespace plus one block. Thus, if the DBA wants to create a tablespace with an extent size of 5120K, and a total useful space of 200MB, and if the block size for the database is 4K, then the following specification should be used ($204804 = 200 * 1024 + 4$)

```
CREATE TABLESPACE foo datafile 'bar' size 204804K;
```

Caution - If you are using raw devices then the raw device size must equal the desired file size plus two blocks. This is because on some platforms an extra block at the beginning of the raw device is needed by Oracle. In the above example the raw device should be sized to be greater than or equal to 204808K, with the rest of the specification staying the same. Yes, we realize that being `SAFE` can be annoying at times.

If the `autoextend` feature is enabled for a file, make sure that the maximum size is a multiple of the extent size plus 1 block. The default value for `AUTOEXTEND NEXT` is one block. Therefore, Oracle will extend files only as much as necessary

to satisfy the extent size. Since extents are uniform, the file will always extend by the same amount. If you explicitly set the AUTOEXTEND NEXT value for a file, then set it to a multiple of the extent size for the tablespace.

2.1.12 Never Defragment the Space Within a Uniform Extent Tablespace

If you follow SAFE, then tablespaces will not become fragmented and therefore will never need to be defragmented. Some administrators are bothered by free extents that occur in the middle of a file. They like to have all free space at the end of the files in one extent. This seems cleaner and neater to them and therefore they will relocate all the segments in the tablespace to achieve this. This is totally unnecessary in the case of uniform extent size tablespaces. You should never reorganize the segments in a uniform extent size tablespace. This is a waste of time, effort, and availability. Even though free space may be spread throughout the tablespace it is no less usable than space at the end of the file. Administrators that reorganize the space in a uniform extent size tablespace incur the risk of making an error for no gain. It is paradoxically true that a tablespace using uniform extent sizes with free space sprinkled all over the tablespace is better managed than a tablespace with multiple extent sizes having all the free space coalesced at the end of the files.

Oracle will not proactively coalesce multiple contiguous free extents into one free extent in tablespaces with PCTINCREASE set to 0. Therefore, if you follow the SAFE rules, you may see many free extents that are adjacent to each other. This is not a problem. Coalescing these extents proactively would add overhead while providing no benefit. As long as the size of all free extents is a multiple of the tablespace extent size, the tablespace is not fragmented. Oracle will automatically coalesce any free extents it needs to satisfy a space allocation as on-demand coalesce has no correlation to PCTINCREASE for the tablespace.

2.2 Scripts for Implementing and Administering SAFE

2.2.1 Rules 1-3: Data Tablespaces should have uniform extent size (160K, 5120K, 160M)

The following syntax can be used to create a tablespace using 5120K uniform extent sizes. Sizing the datafile is a little tricky since it must be specified in terms of K instead of M to add the extra header block. In this case we want a file of size 100M plus a 4K header block so we use $1024*100+4 = 102404K$. The header block size is the same as the block size for your database. We also do not wish the file to grow beyond 200M in size.

```
CREATE TABLESPACE    tbspc_name
DATAFILE 'data_filename' size 102404K AUTOEXTEND ON MAXSIZE 204804K
MINIMUM EXTENT 4M
    DEFAULT STORAGE (INITIAL 4M NEXT 4M PCTINCREASE 0
                    MINEXTENTS 1 MAXEXTENTS 4096);
```

Having MAXEXTENTS set to 4096 avoids potential problems which might occur if an object grows unexpectedly and ends up with 10s of thousands of extents. It also allows for a significant number of extents above the targeted maximum preventing segments from becoming unavailable due to minor mistakes or time constraints.

To check for Data tablespaces that do not obey the uniform extent size rules you can run the following query:

```
select tablespace_name, initial_extent, next_extent, pct_increase, min_extlen
from dba_tablespaces
where (initial_extent not in (160*1024, 5120*1024, 160*1024*1024)
or next_extent    != initial_extent
or pct_increase  != 0
or min_extlen    != initial_extent)
and contents = 'PERMANENT'
and tablespace_name != 'SYSTEM'
and tablespace_name not in (select tablespace_name from dba_rollback_segs);
```

2.2.2 Rule 4: Monitor and Potentially Relocate Segments Having More Than 1024 Extents

To check for segments having more than 1024 extents you can run the following query:

```
select owner, segment_name, extents from dba_segments
       where extents > 1024 and segment_type != 'TEMPORARY';
```

2.2.3 Rule 5: The Maximum Single Segment Size Should be Somewhere between 4G and 128G

To check for segments that are larger than 4G you can run the following query:

```
select owner, segment_name, bytes from dba_segments
       where bytes > 4*1024*1024*1024 and segment_type != 'TEMPORARY';
```

2.2.4 Rule 6: Very Large Tables and Indexes should be Placed in a Private Tablespace

To find tables or indexes that are larger than 4G you can run the following query:

```
select owner, segment_name, sum(bytes) from dba_segments
       group by owner, segment_name
       having sum(bytes) > 4*1024*1024*1024;
```

2.2.5 Rule 7: Temp Segments should be restricted to TEMP tablespaces

To identify users that do not have their TEMP TABLESPACE set to a tablespace of type TEMP you can run the following query:

```
select username from dba_users, dba_tablespaces
       where temporary_tablespace = tablespace_name
       and contents != 'TEMPORARY';
```

2.2.6 Rule 8: Place Rollback Segments in Tablespaces Dedicated to Rollback Segments

To identify tablespaces which have both rollback segments and user data you can run the following query:

```
select tablespace_name from dba_segments
       where segment_type != 'ROLLBACK'
       and tablespace_name != 'SYSTEM'
       and tablespace_name in (select tablespace_name from dba_rollback_segs);
```

2.2.7 Rule 9: TEMP and UNDO Tablespaces should contain between 1024 and 4096 extents

To check for UNDO and TEMP tablespaces that do not obey the extent size rules you can run the following query:

```
select tablespace_name, initial_extent, next_extent, pct_increase, min_extlen
       from dba_tablespaces t,
       (select tablespace_name tbspc,
              sum(bytes) tbspc_sz,
              count(*) num_files,
              sum(bytes)/sum(blocks) blk_sz
       from dba_data_files
       group by tablespace_name) f
       where
       ( initial_extent < (tbspc_sz - blk_sz * num_files) / 4096
       or initial_extent > (tbspc_sz - blk_sz * num_files) / 1024
       or next_extent != initial_extent
       or pct_increase != 0
       or min_extlen != initial_extent)
       and tablespace_name = tbspc
       and tablespace_name != 'SYSTEM'
```

```
and ( contents = 'TEMPORARY'
      or tablespace_name in (select tablespace_name from dba_rollback_segs));
```

2.2.8 Rule 10: Never Place User Data in the System Tablespace

To identify user data in the System Tablespace you can run the following query:

```
select owner, segment_name from dba_segments
where tablespace_name = 'SYSTEM'
and owner != 'SYS'
and owner != 'SYSTEM';
```

2.2.9 Rule 11: Datafile Size should be a multiple of extent size + 1

To check for datafiles that do not obey the file size rules you can run the following query:

```
select t.tablespace_name, file_name, bytes file_size, initial_extent
from dba_tablespaces t, dba_data_files f
where mod(f.bytes - f.bytes/f.blocks, initial_extent) != 0
and f.tablespace_name = t.tablespace_name;
```

2.2.10 Rule 12: Never Defragment the Space Within a Uniform Extent Tablespace

To check for a tablespace using a uniform extent size that has a free extent that is not a multiple of the extent size you can run the following query:

```
select t.tablespace_name, file_id, block_id, bytes, initial_extent
from dba_tablespaces t, dba_free_space s
where next_extent = initial_extent
and pct_increase = 0
and min_extlen = initial_extent
and t.tablespace_name != 'SYSTEM'
and t.tablespace_name = s.tablespace_name
and mod(bytes, initial_extent) != 0;
```

3. Heap (aka Table) Segments -

Oracle allows tables to be stored in Heaps or in B+Trees. Fragmentation issues for B+Trees are similar whether they are storing index or table data and we discuss B+Tree related issues in the next section. A heap as the name suggests is managed as a collection of blocks and space is consumed from these blocks as needed, for inserts and updates. Rows themselves can be of varying sizes and have to be placed appropriately in the blocks belonging to the heap. Over time as rows get deleted/updated/inserted, a similar problem as discussed for tablespaces can occur within the segment. And since the row sizes cannot be required to be fixed, we need to come up with other somewhat more complicated means of avoiding fragmentation for heap segments.

The algorithm which Oracle uses to place rows in a segment, optimizes for time (for access). Hence, we try to place the rows in the minimum number of blocks necessary to hold them. Based on past history of the segment, there might be some free space in each of the blocks but not sufficient space in any one of them for Oracle to be able to insert a new row. This will cause the segment to grow, i.e. allocate another extent from the tablespace. This problem is referred to as heap fragmentation.

3.1 Heap Freelist Management -

Oracle manages free blocks as a singly linked list based on the the following parameters -

- **PCTFREE** - the percent of space to leave in the block for updates. Thus, if a block is on the freelist and say pctfree is 10 then we shall insert data into it as long as the used portion of the block is less than 90%. This parameter does not apply to updates which cause a row in the block to grow, even if the block is not on the freelist. The default for this is 10.
- **PCTUSED** - The minimum percent of space we want used in every block. If a block is taken off the freelist and subsequent deletes cause the percentage of used space in the block to go below pctused, then it will be placed back on the free list. This makes it a candidate for use by subsequent inserts. The default for this is 40.

To further illustrate free space management, lets look at the following example with Scott's EMP table in a database with a block size of 4K.

```
CREATE TABLE EMP
  (EMPNO NUMBER(4) NOT NULL,
   ENAME VARCHAR2(10),
   JOB VARCHAR2(9),
   MGR NUMBER(4),
   HIREDATE DATE,
   SAL NUMBER(7,2),
   COMM NUMBER(7,2),
   DEPTNO NUMBER(2));
```

Oracle stores numbers in a portable varying width format on disk. Varchars, by definition store as many characters on disk as are specified for the given column. The maximum row size (including overhead) for the above given schema is 57 bytes. The interested reader can refer to Oracle8 Concepts as well as Oracle8 Administration guide for a detailed explanation of how we calculated this size.

3.1.1 Why and How is PCTFREE relevant ?

PCTFREE is used only if the rows are getting updated. Oracle does not change the ROWID of a row for its lifetime. Thus, if there is not enough space in the block for a given row to be updated, then the row gets migrated to another block (or blocks). However, a forwarding piece is left in the old position to maintain a fixed rowid.

Hence, rowid based probes will result in 2 block accesses in these situations if PCTFREE is not set properly. Note that this does not really cause significant space under-utilization as the forwarding piece is only about 10 bytes long.

For Scott's EMP table, it is not likely that an update will change the row size much, though it definitely could as the job title, salary, department etc. could change, and each of these are stored in varying width columns. However, a closer look at the expected sizes for these columns shows that an average of 10% increase in size is probably good enough for these rows. Thus leaving 10% for updates is advisable, hence PCTFREE should be set to 10%.

Note that being too pessimistic on PCTFREE can waste some space but will more or less guarantee no row-chaining while being too optimistic can result in row-chaining (reduced performance). Instead of trying to save every byte in a block (it just wont happen!) we recommend erring on the side of setting PCTFREE high rather than low as in the most cases this probably will be only a difference of about 5% which will definitely not all get wasted. For tables which dont have varying width columns and/or dont get updated, this parameter can be safely ignored or set to a really low value.

3.1.2 Why and How is PCTUSED relevant ?

Blocks are put on the freelist when the utilization goes below PCTUSED. Subsequently, we insert more rows into the given block till it reaches PCTFREE and then we take it off the free list.

For Scott's company assuming employees dont leave as often, it probably wont matter much what exactly this number is set to, but lets look at what it shouldnt be set to. Since we put a block on the freelist as soon as it goes below PCTUSED, it

is necessary that $\text{blocksize} * (100 - \text{PCTFREE} - \text{PCTUSED})$ be able to hold at least a row, else putting it on the list is useless as we won't be able to use it. Since our max row size = 57 bytes and block size = 4K - overhead (~100 bytes), $(100 - \text{PCTFREE} - \text{PCTUSED}) \sim \geq 2$, i.e. PCTUSED should be at most 88.

What if we have a block at the head of the free list and can no longer use it? This can happen as the row being inserted can be bigger than the amount of space available in the block (remember that row sizes are not fixed).

What do we do in this case? We have a few choices -

1. Remove it from the free list - we do so if the block is above PCTUSED.
2. Walk the list i.e. go to the next block on the list - We do not do so as it is not clear for how long this problem block will persist at the head of the list and as it also slows down the search.
3. We grab more blocks from the segment and put them on the freelist - We do this if we cannot do (1).

Note that (1) above uses PCTUSED to decide whether a block can be taken off the freelist even though it is not filled all the way to $100 - \text{PCTFREE}$. If we can't take it off the freelist then we grow the heap (segment).

Hence, if the row sizes vary a lot then setting PCTUSED to be too high can cause space growth and thus this parameter should be set with care. It is better to work with the larger row sizes than the average row sizes when trying to estimate PCTUSED. If a good estimate cannot be made of the row sizes, it is again better to err on setting pctused to a small value as it has the benefit of letting the heap reach a steady state size (which may not necessarily have great space utilization, but is better than a heap which keeps growing). Also, if the row sizes are not big then the difference between a safe and an aggressive choice is not much and space wastage again will be bounded with reasonable limits.

We believe that if the row sizes are not big then the default value of PCTUSED should work quite well. If row sizes are big i.e. the maximum row size can be larger than half a block then we recommend setting PCTUSED to 10.

Note that having a small pctused (which is a safe choice) has a detrimental effect on space utilization ONLY when rows are being deleted at random from the table, and not all rows will get deleted over time, in which case it may take a long time before a block goes below pctused and hence gets reused. For this class of applications if the steady state space utilization of the heap ends up being low then they should take a closer look at tuning PCTUSED.

The above can be summarized using the following formula -

$$\text{PCTUSED} = 100 - \text{PCTFREE} - \max(10, (\text{maximum-row-size}/\text{blocksize}) * 100);$$

If the above ends up being negative then PCTUSED should be set to 1.

3.2 Various Access Patterns on Heaps -

1. Mostly Insert Only - Have a small PCTFREE, leave PCTUSED alone.
2. Queue like - Table is used as a queue where rows get inserted and subsequently get deleted soon after - Small PCTFREE, very small PCTUSED.
3. Random Insert/Delete - Estimate row-sizes and follow above recommendation.
4. Mostly Update - PCTFREE based on how much the row size is expected to change. leave PCTUSED alone.
5. Other combinations - infer from above.

3.3 Monitoring Heap Space utilization and row chaining -

Oracle provides various statistics on heap space utilization, and these can be used to tune the setting of PCTFREE and PCTUSED for a given heap segment. Note that these views do not provide history of DML activity, and hence should be used in conjunction with some idea of DML pattern on the table to tune the parameters.

The following columns of DBA_TABLES and DBA_TAB_PARTITIONS can be used to get statistics on heaps after doing an ANALYZE TABLE ESTIMATE/COMPUTE STATISTICS on the heap(s).

```

NUM_ROWS      - no of rows
AVG_SPACE     - average available free space per block
CHAIN_CNT     - no of chained rows
AVG_ROW_LEN   - average length of a row

```

1. The average length of the row should be used to look at PCTUSED and PCTFREE and verify if $(100 - (PCTFREE + PCTUSED)) * \text{blocksize} > \text{avg_row_len}$.
2. $\text{chain_cnt}/\text{num_rows}$ gives the pct of rows which are chained. If this number is high (more than 5 percent) and maximum row-length is $<$ block size then PCTFREE should be definitely increased. If the pct of chained rows is small (less than 1 percent), and maximum row-length is $<$ block size, then PCTFREE should be increased only if maximum row-length does not vary much from avg_row_len , else space will be wasted.
3. $(\text{blocksize} - \text{avg_space})/\text{blocksize}$ gives space utilization. If rows are not big, then decreasing PCTFREE and increasing PCTUSED will improve space utilization. Be careful when increasing PCTUSED and always err on the side of a lower PCTUSED, i.e. use maximum row size.
4. If maximum row-length $>$ blocksize then reverify that PCTUSED is set to a very small value. If avg_row_len is much less than block size but the maximum is larger, then space utilization may be low if rows are getting randomly deleted. This cannot be avoided but by doing a rebuild. Setting PCTUSED to a larger value may help a bit but can cause the space utilization to go even lower based on what the size distribution looks like which is difficult to figure out. Hence it is recommended that PCTUSED be kept low at the expense of a larger steady state heap size with periodic rebuilds.

4. Index (B+Tree) Management

As discussed in the previous section, tablespaces can get fragmented due to the algorithms and data structures used to managed the space. B+Trees on the other hand have a more rigid structural requirement and can get fragmented due to repeated manipulations of this structure (the somewhat balanced tree).

The parameters which affect block management of index segments are the following -

1. INITRANS - number of ITLs to have in a given leaf block. This number should be equal to expected concurrency on any given leaf block. Note that Oracle grows ITLs in a block if the concurrency increases and there is space available in the block.
2. MAXTRANS - maximum number of ITLs to allow in a leaf block. This is useful in preventing excessive number of ITLs in a leaf block due to momentary spikes in concurrency. This is relevant because on an index leaf block split, we format the new leaf block being created with the same number of ITLs as in the block being split.

The above two ITL related parameter should be pretty much always left at the default values.

3. PCTFREE - amount of free space (expressed as a pct of the block) to leave unused (free) in the block at index create time. This is useful when the index is being created on a populated table and subsequent inserts in the index are expected to be random.

A leaf block is split when an incoming key (and associated data) cannot fit in the relevant leaf block. The choice of split point is based on the leaf block being split. If the key being inserted is at the right-most end of the tree, then the split basically creates a new leaf block and puts the new key into it. This is referred to as the 99-1 split. If the key is anywhere else in the tree, then we do an approximate 50-50 split, i.e. it is assumed that modifications to the tree are random and we thus create two roughly equal leaf blocks. For Index-Organized Tables introduced in Oracle8, the leaf is typically split at the insert point.

On Deletes from the tree, if a leaf block becomes empty, then it is put on the list of free blocks. However, it is not removed from the tree at this point. Removing the block from the tree before the active transactions in it have committed is very expensive if not impossible. If it is a random insert/delete application then there is a reasonable chance that this block will have a key inserted into it soon and hence it is not worthwhile spending time to take it off the tree immediately. Later, when the block is reused for a different portion of the tree it is removed from the old position and placed in the new one.

4.1 Typical B+Tree modification patterns

Let us look at different access patterns to the Index and the expected behaviour of the tree.

4.1.1 Insert Only - Right Growing -

If the index is mostly insert only and the keys are inserted in ascending order towards the end of the tree, we call it a right-growing tree. This would be typical for an application which uses some form of sequence numbers to tag records and indexes on this sequence number. In this situation, the space utilization of the tree would be very high and reorgs will not help any.

4.1.2 Insert Only - SubTree Right Growing -

If the index is mostly insert only and the keys are inserted in ascending order towards the end of multiple sub-trees, we call it a subtree-right-growing tree. This would be typical for an application which prefixes the primary key with some kind of number to partition the data and the primary key uses some form of sequence number. In this situation, since Oracle will tend to do a 50-50 split (for indexes, not IOTs which do insert-point split), over time the space utilization will go down and the index will most probably become a candidate for reorganization.

4.1.3 Queue like Indexes -

Insert on right and delete from left - This would again be an application which uses sequences to generate tags, but the data being inserted into the base table is being processed like a queue. Inserts will come in on the right and be deleted from the left (FIFO). Since Oracle reuses blocks which become empty, the blocks as they become empty on the left side of the tree will be reused on the right end. Note that if there is a massive purge i.e. delete from the tree, then till enough inserts are done, it would seem like the tree has lot of empty blocks. This is because of the delayed removal of empty blocks from the tree. In a steady state queue, this will never happen.

4.1.4 Mostly Insert - Random -

This would be the case where the index is on a column of a table which is never updated and the rows from the table are deleted infrequently. In this case, the 50-50 algorithm discussed above would come into play and the utilization of the index blocks should be > 50% in steady state. Note that the tree may start off being 100% utilized (if it is built on a pre loaded table) but as rows get inserted, the utilization will eventually go down. Analysis found in standard database texts show that the average utilization of the index is expected to be about 66%.

Since the average utilization is expected to be 66% it is unlikely that rebuilding the index is going to reduce a level and will provide momentary compaction in any case (which the random insert activity any way is doing for us).

Hence, the only case for rebuild probably is if the table was a mostly insert only table and has now become read-only for some reason. In this case a tree rebuild will free up space and provide better compaction for leaf blocks and hence better range scan performance.

The above example assumed that there is high (and random) insert activity on the table. If the insert activity is not expected to be high but random, then at the time of building the index (which is presumably being built on a pre-loaded

table), Oracle recommends setting the PCTFREE parameter to reduce the pct of the block filled at create time so that subsequent inserts will not cause any potential utilization problems.

4.1.5 Insert and Delete - All Random -

This probably is common for secondary indexes on volatile columns of tables. As mentioned previously, Oracle does not compact (coalesce) non-empty leaf blocks. Since the insert/delete activity is random, in steady state the tree should be at a 66% utilization like the case discussed above. However, if the modifications to the index are skewed, then the tree may get fragmented and need to be rebuilt.

Secondary indexes on queue like tables will also end up with having a low % utilization over time, though they will tend to achieve a steady state probably higher than 66%.

When the utilization of the tree is really bad and rebuilding it can reduce a level of the tree, then it may be worthwhile to rebuild the index. If the tree is not going to reduce by a level, then unless the application is doing big range scans, it is not worthwhile to rebuild the index. The space reclaimed from rebuilding the index in either case is probably not going to be too much (as a percentage of the the db size) and should not be the determining factor in rebuilding the index. The only other case when it might be worthwhile to rebuild indexes even without reducing a level is when the index is hot and tends to stay all cached (including all leaves) in memory (the buffer cache). In this situation, even though disk savings would be minimal, buffer cache savings could be substantial and may validate reorganizing the index every so often.

4.2 Monitoring Index Fragmentation

Since the potential savings from reorganizing the index is strongly correlated to its usage, it is difficult to come up with a set of simple rules to enable the decision of index reorganization. As we have also noticed it is more the performance gains rather than the disk savings which warrant reorganizing (or defragmenting) an index. Instead of going into details of when to rebuild an index, we provide information about views which aid in determining tree utilization and refer the user to other performance texts to determine when the reorganization will provide performance gains worth the overhead involved.

The views which provide various statistics about indexes are the following -

1. DBA_INDEXES - for non partitioned indexes and aggregate information for all the partitions.
2. DBA_IND_PARTITIONS - identical to above view, but provides information about individual partitions.

The columns of interest from a fragmentation perspective in these views are -

```
BLEVEL - the btree level - 0 based
LEAF_BLOCKS - Number of leaf blocks in the tree
NUM_ROWS - number of rows in the tree
SAMPLE_SIZE - what sampling did we do - provides level of confidence
```

Unfortunately, the average row size is not provided as a part of this view. The user would have to thus calculate the average row size based on the columns in the index and use that to get utilization factor from the above view.

```
utilization_factor =
(NUM_ROWS*average_row_size)/(LEAF_BLOCKS*usable_block_size) where
usable_block_size = database block size - standard overhead (~ <= 100 bytes).
```

There is another view which provided more detailed information about the structure of the tree but requires that the DBA do an INDEX VALIDATE STRUCTURE. An INDEX VALIDATE STRUCTURE causes the table to become read-only and thus can be only done during a planned down time. The view which provides information about the index after doing a VALIDATE STRUCTURE is called INDEX_STATS.

Some interesting columns in this view are

```

br_rows          - number of rows in all the branch blocks
br_rows_len      - sum of the lengths of all the rows in branch blocks
br_blks          - branch blocks
br_blk_len       - usable space in a branch block (block size - overhead)
lf_blk_len       - usable space in a leaf block (block size - overhead)
lf_rows         - number of leaf rows
lf_rows_len      - sum of the lengths of all the rows in leaf blocks
del_lf_rows      - number of deleted rows still present in leaf blocks
del_lf_rows_len  - sum of the lengths of all deleted rows in leaf blocks
height          - current height of the tree

```

We here reproduce the formula existent in most database textbooks on how to calculate expected optimal height of the tree based on the statistics provided by the above view.

Let $rows_per_leaf_block = lf_blk_len / ((lf_rows_len - del_lf_rows_len) / (lf_rows - del_lf_rows))$ be the expected 100% utilization of a leaf block.

Even though the number of rows in all branch blocks is not the same and this number typically tends to increase as we get closer to the root block, the following is a reasonable estimate of the expected fanout of the branch block. Let $fanout = br_blk_len / (br_rows_len / br_rows)$

The \log (base fanout) $((lf_rows - del_lf_rows) / rows_per_leaf_blocks)$ is the expected number of branch levels. This plus 1 provides the optimal (based on many average assumptions) height of the tree. Comparing this to height of tree from above view tells us whether we shall reduce a level. Transforming this to a SQL query which tells us whether a level will be saved as well as the utilization factor -

```

SELECT name NAME, partition_name PARTITION_NAME,
       (br_rows_len*100)/(br_blk_len*br_blks) BRANCH_UTILIZATION,
       ((lf_rows_len - del_lf_rows_len)*100)/(lf_blk_len*lf_blks) LEAF_UTILIZATION,
       decode (SIGN(ceil(log(br_blk_len/(br_rows_len/br_rows),
                          lf_blk_len/((lf_rows_len - del_lf_rows_len)/(lf_rows - del_lf_rows))))
              + 1 - height), -1, 'YES', 'NO') CAN_REDUCE_LEVEL
from INDEX_STATS;

```

5. Using Partitions to Eliminate Fragmentation

Many systems don't perform any online delete operations. Instead, data is kept online to serve as history even after it is no longer actively used. In this case, old data is periodically deleted in bulk from the database.

Oracle8 partitions provide a means of deleting the old data in a very space and time efficient manner. If a date column or sequence number is used as the partitioning key for a table, the oldest partition of the table can be periodically dropped instead of performing the delete on the old rows. This eliminates the runtime overhead of the delete operations and leaves the remaining rows tightly packed with no fragmentation. This also has the added benefit of grouping the most recent rows close together which tends to improve caching behavior.

When indexes exist on the table, the indexes can also be partitioned on the same date column as the table. This type of index is known as a local index. When the oldest table partition is dropped, the corresponding index partitions will also be dropped.

The disadvantage of this scheme is that index lookups will be slowed down since index operations will have to probe all the index partitions. This may increase overhead significantly for queries that would normally access a small number of rows.

If this is a concern then it is best to keep the number of partitions small and make sure most queries restrict the date range (partition key value) so that it matches as few partitions as possible.

6. Conclusion

In this paper, we provided an overview of the various kinds of fragmentation which can occur in Oracle and the techniques to avoid, detect and rectify them. We discussed common policies which can be deployed to avoid fragmentation for most of the cases. Though fragmentation is not necessarily always avoidable the authors believe that this subject has been overly exposed and the time spent on this matter could probably be well spent otherwise. We encourage Oracle DBAs to further refine this document with their own policies and techniques. We would also like to thank Jonathan Klein, Franco Putzolu, Alex Tsukerman and Richard Sarwal amongst others for kindly taking the time to review this document and helping making it accurate and hopefully complete.