dbazine.com

bmcsoftware

ORACLE Index Management Secrets:

Top Oracle Experts Discuss Index Management Techniques

with a foreword by Don Burleson

Donald Burleson
Jonathan Lewis
Mike Hordila
John Weeg
Daniel Liu
Lisa Hernandez

ISBN: 0-9740716-9-2
Retail Price $14.95 US/$22.95 Canada

RAMPANT
TECHPRESS
eBook

Oracle Index Management Secrets

Top Oracle Experts Discuss Index Management Techniques

Donald K. Burleson
Jonathan Lewis
Mike Hordila
John Weeg
Daniel Liu
Lisa Hernandez

RAMPANT
TECHPRESS

Oracle Index Management Secrets

Top Oracle Experts Discuss Index Management Techniques

By Donald Burleson, Jonathan Lewis, Mike Hordila, John Weeg, Daniel Liu & Lisa Hernandez

Printed in the United States of America.

Series Editor: Donald K. Burleson

Production Manager:: John Lavender

Production Editor: Teri Wade

Cover Design: Bryan Hoff

Printing History:

July 2003 for First Edition

The information provided by the authors of this work is believed to be accurate and reliable, but because of the possibility of human error by our authors and staff, BMC Software, DBAZine and Rampant TechPress cannot guarantee the accuracy or completeness of any information included in this work and is not responsible for any errors, omissions or inaccurate results obtained from the use of information or scripts in this work.

Links to external sites are subject to change; dbazine.com, BMC Software and Rampant TechPress do not control or endorse the content of these external web sites, and are not responsible for their content.

ISBN: 0-9740716-9-2

Table of Contents

Conventions Used in this Book

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

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

Hence, all Rampant TechPress books follow these conventions:

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

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

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

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

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

Foreword

By Donald K. Burleson

Over the past decade, Oracle Corporation has introduced an unprecedented array of new indexing structures. Stemming from original B*tree index structure, Oracle has progressively introduced more complex index structures to allow Oracle SQL to efficiently access data.

In Oracle8 we saw the addition of bitmap indexes, function-based indexes, as well as reverse key indexes and star index structures. We also saw the Oracle SQL optimizer becoming more intelligent about the way indexes are used by SQL statements. In Oracle9i, we have the index skip scan, the and_equal hint to combine individual indexes into a contaminated index, as well as specialized bitmap access methods to improve the speed of data warehouse queries.

Because indexes are the single most important access method for SQL statements, it is a great benefit to the Oracle professional to understand the internals of indexing. That is what this book is all about. We've collected some of the finest works by some of the finest authors in the world to provide you with detailed information about how to use Oracle indexing most effectively in your production environments.

It is our hope that this short text will provide you with the information you need to select the most appropriate indexes for your SQL queries, and at the same time understand the sophisticated SQL optimization methods that are going on under the covers within your Oracle database.

Regards,

D Burleson

Donald K. Burleson

Automated Table/Index Reorganization In Oracle8i

By Mike Hordila
(View this article, downloadable scripts, and links, online at
http://www.dbazine.com/hordila1.html)

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

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

This PL/SQL package is a complementing solution to the one presented in my article, "Setting Up an Automated Index-Rebuilding System" (Oracle Publishing Online - September 2001). It can be run as a periodic complement to the auto-reindexing package (PKG_NDXSYS), or instead of it. This solution has been tested on Unix (HP 10.7 and 11, Sun Solaris 7 and 8, AIX 4.3, Linux 2.x) and Windows servers (NT4, 2000), on Oracle versions 8.1.5, 8.1.6, 8.1.7. It should work just fine with Oracle9i, but not with versions earlier than 8.1.5. It requires some knowledge of UNIX shell scripts, SQLPlus scripts, and PL/SQL.

However, the full scripts are provided and minimal knowledge would be enough to install the package and get started.

When Reorganizing, How Many Extents to Use?

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

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

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

Possible Reorganizing Strategies

- Cron jobs at fixed times — the most common strategy.
- Inside batches, after massive changes, to keep objects current — a fairly common strategy.
- Inside batches, before massive changes, to improve the batch performance — used less frequently.
- Just before backups, to back up an optimized database, or just after backups, to back up the database faster.
- Dedicated systems, with collection tables, control procedures, and so on — used in more complex environments.

Assumptions and Experimental Figures

- On average, roughly, we could rebuild in one hour 5,000,000 rows or 5GB.
- We had a time window of between one and three hours, between 21:00 and 24:00.

- We could not do all objects in one day (session).
- Time acceptable without reorg is within a certain limit.
- Some days cannot be used for reorg, as they are being used for cold backup, and so on.

Some Procedures Related to Table Reorganization

- Coalesce free extents in tablespaces, before and after each reorg; here is a script: *ts_coalesce.sql*
- Build a fragmented tablespace as a test environment; use a script such as: *ts_fragment.sql* (first create a user "cbosys2" and a tablespace for it)
- The only visible objects in a GUI tablespace tool like Tablespace Manager (Map) are the ones that actually take up physical space: tables and indexes (regular, primary keys, unique constraints, and so on). If some objects are not visible, it means they are just references or definitions (foreign keys, not nulls, checks, and so on).
- Determine the fragmentation level in a database — here are some fragmentation assessment criteria:
    - high numbers of extents (acceptable < 1024 extents for very large objects — look out for extents per object > 5)
    - high percentages of chained rows per table (acceptable < 3 percent — look out for percentages > 0.1 percent); analyze the tables first
    - high percentages of free space inside blocks (look out for FREESPACE/BLOCK > 2*PCTFREE)
    - high percentages of free space above high watermark (look out for EMPTY BLOCKS ABOVE HWM > 50 percent)
- here are a few scripts to help with these tests: *objects_for_reorg.sql* and *obj_next_ext_fail.sql*

Important Issues Regarding Table/Index Moving/Rebuilding

- You cannot perform a table reorg without an index reorg, even if you do not re-structure or relocate the index, because the index becomes UNUSABLE after the table reorg, as the ROWID references become invalid. That is why, if you have a table reorg scheduled, you may skip a scheduled index reorg (for the affected indexes).
- The new index is built either from the data in the index, or from the data in the table, whichever source is smaller. This is called "fast rebuild" and is available since Oracle 7.3.4. If you suspect the index is already corrupted, you will have to drop the index and re-create it with fresh data from the table.
- Oracle will place a lock on the table for the duration of the table/index move/rebuild. The lock affects INSERT, UPDATE, DELETE statements, but allows SELECT statements. The DML will have to wait until the move/rebuild is done. However, Oracle8i (8.1.x) can allow any DML statement if the DDL runs the ONLINE option.

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

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

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

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

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

Limitations of the "`ALTER TABLE MOVE`" Command

- Supported only in Oracle8.1.5 and higher.
- Does not support directly some objects and some data types:
    - Clustered tables, IOTs, overflow table of an IOT, hash and composite partitions (range partitions are supported), tables with columns containing LONG and LONGRAW types, tables with columns containing user-defined types, indexes on such columns, function-based indexes, domain indexes. Partitioned tables containing a LOB column can be reorg'ed on a per partition basis only; partitioned indexes may not be rebuilt as a whole. For some object types and data types, there are special commands that can be used as workarounds.
- Most of these limitations apply also to Create Table As Select (CTAS) methods.
- Some of them apply also to index rebuilds.
- You can still use the `SQL*Plus COPY` command or the EXPORT/IMPORT utilities.

Manual Object Reorganization

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

- *reorg.sql* - script to reorg all tables in the database
- *reindex.sql* - script to rebuild invalidated indexes - called by *reorg.sql*
- *ts_coalesce.sql* - script to coalesce tablespaces - called by *reorg.sql*

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

Step 1

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

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

Step 2

The table move will change the ROWIDs of the table rows, and as such the indexes, which are based on ROWIDs, will become invalid

(UNUSABLE). Therefore, step two must be executed immediately after step one: rebuild the invalid indexes on the current table.

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

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

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

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

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

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

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

Automated Object Reorganization

Our strategy will be a combination of cron jobs and a PL/SQL package (PKG_TABSYS).

Reorganizing tables/indexes normally can be done online, without dropping objects, and has a very positive impact on the general performance of the database. I have been running the package for the last year with no serious problems. The execution times seem to decrease steadily after a few runs, as the package has some self-tuning capability. The average move/rebuild times on Oracle8i have come down from 90 minutes to 45 minutes. In theory, at least, the more it runs, the less fragmented the objects become, and the faster the systems will be. Some degree of tablespace level fragmentation is to be expected.

Remember that tablespace fragmentation does not affect performance, but only the growth capacity of the objects (especially very large ones). You should keep an eye on the free space.

Prerequisites

Before you can begin, you should have some system privileges (see the beginning of the *INSTALL_TABSYS.SQL* script).

- Set *utl_file_dir* = * (or at least c:\temp, or /tmp, etc.) in *init.ora*, to allow log files to be created.
- Set *job_queue_processes* = 2 (or higher) in *init.ora*, to allow *dbms_job* scheduling to work.

Associated Tables

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

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

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

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

Overview of the Package

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

- clean up residual temporary segments
- coalesce free space in tablespaces
- analyze the structural integrity of the objects

- generate valuable statistics usable by the CBO
- de-allocate unused space from object blocks
- shrink object segments
- realign the high watermark to low levels
- reorganize fragmented objects into fewer extents
- restructure (optimize) tablespace storage options
- restructure (optimize) table storage options
- compact table blocks into fewer blocks
- reattempt to run with modified parameters in case of failure
- generate alerts if it detects failure to grow or reorg
- detect some generic unavailability conditions
- process both tables and indexes
- reorganize/defragment, actually, the entire database

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

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

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

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

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

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

Setup

The package is installed into the default Oracle schema MHSYS, which I use to host my automation packages. *It can be installed, as is, for UNIX and NT-based servers.* It is a pretty comprehensive piece of software, is compatible with Oracle8.1.5 or higher, on both UNIX and NT, and includes routines to detect the current OS, Oracle version, and SID.

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

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

About the Author

Mike Hordila is an OCP v7-8-8i, and has his own consulting company, DBActions Inc. Mike is also the author of "Automated Cost-Based Optimizer" (Oracle Magazine Online — September 2000) and Setting Up an Automated Index-Rebuilding System (Oracle Publishing Online — September 2001). Updated versions of his scripts and packages are available on his web site. Mike is also an Oracle technical editor with Hungry Minds (formerly IDG Books). Read more of his articles at http://www.dbazine.com.

Locally Managed Indexes

By John Weeg

(View this article, downloadable scripts, and links, online at http://www.dbazine.com/weeg7.html)

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

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

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

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

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

Rebuild in the Same Tablespace

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

```
Ext1(128k),other index,ext2(64k),other index, ext3(64k), other index
```

If you leave the index definition as it is, the rebuild will recreate the index in the first block that can hold 128k, resulting in:

```
Ext1(128k),other index,ext2(64k),other index, ext(64)3, other
index,temporary(128k)
```

and then:

```
free(128k),other index,free(64k),other index, free(64)3, other
index,ext1(128k)
```

Now there is more free space mixed in with the indexes, and if the index grows and can't fit in 128k anymore, you may end up with chunks of free space that are unusable.

No Fragment

To avoid this fragmentation, the common approach is to rebuild all of the indexes in the tablespace into another tablespace, coalesce this tablespace, and then rebuild them back. This means rebuilding the index twice when you want to do it once.

The other option is simply to drop the indexes, coalesce the tablespace, then recreate. This will set any objects depending on this table to an invalid state and they will need to be recompiled. Depending on sizes, it is usually faster to rebuild.

8.1 to the Rescue

To avoid spending the time to rebuild, you should ensure that all extents in the tablespace are the same — "initial" is the same as "next" and all indexes have the same ... what? Then it doesn't matter if the tablespace becomes fragmented because all the space remains usable. If you are going to do this, you should also take advantage of the new locally managed tablespaces that Oracle provides in V.8.1.

First, create a tablespace and give it a uniform extent size:

```
Create tablespace local64k_idx
Datafile '…/local64k_idx01.dbf' size 512M
Autoextend on next 10M maxsize unlimited
Extent management local uniform size 64k;
```

Next, put the indexes in this tablespace and don't worry about fragmentation.

Now before you start thinking, "finally, this guy wrote a short article," here's another important question: When should you decide to rebuild an index and reclaim the empty space within it? I usually say that an index that is in more than four extents should be rebuilt. And what if the index is really 1M? Should you rebuild it each time when a rebuild is not needed at all?

More Than One

You probably already know the answer to that question. You will simply have multiple tablespaces, each locally managed at different sizes. Since my tolerance is an index in four extents, I create one tablespace at an extent size of two blocks, one at eight, one at 32, and one at 128. See how this all falls into my four's? If I have an 8k block size, then I create a 16k, 64k, 256k, 1M.

So where do you put what? Of course, you have to start with a guess. Go ahead and put them in whichever of the four extents you think is correct, and analyze all of them. The rebuild script will put each where it belongs.

What Goes Where

The idea of the rebuild is that any index that is between the extent size for this tablespace and the extent size for the next tablespace belongs in this tablespace. You should pull all of these indexes into this tablespace. So, we have the following:

Tablespace Extent	Index Size
16K	indexes less than 64k
64K	indexes >= 64k and less than 256k
256K	indexes >=256k and less than 1M
1M	indexes >=1M

Break Points

So as not to be fooled by over-allocated indexes, you should check the *leaf_blocks* for the index instead of the bytes. This gives a true picture of space used instead of space allocated.

Assuming you have a block size of 8k, you should first find the number of blocks in 64k to use as your comparison point.

```
variable limit number
begin
select 65536/value into :limit
from v$parameter where name = 'db_block_size';
end;
/
print :limit
```

Script

Each tablespace will have its own script, but they are all basically the same, as indicated by the following:

```
spool rebuild_local16.sql
select 'alter index '||owner||'.'||index_name||' rebuild' ||chr(10)
'tablespace local16k_idx'||
' nologging;'||chr(10)||
'analyze index '||owner||'.'||index_name||' compute statistics;'from
dba_indexes
where leaf_blocks < :limit
and owner not in ('SYS','SYSTEM')
and last_analyzed is not null
and partitioned= 'NO'
and tablespace_name != 'LOCAL16K_IDX';
spool off
@rebuild_local16.sql
```

For the other tablespaces, use the following `where` clauses:

```
64k: where leaf_blocks >= (:limit) and leaf_blocks < (4*:limit)
256k: where leaf_blocks >= (4*:limit) and leaf_blocks < (16*:limit)
1M: where leaf_blocks >= (16*:limit)
```

See the pattern?

Each tablespace will pull in all the indexes that belong in it. If you have partitioned indexes, just throw in a union with *dba_ind_partitions*.

Note that you are only analyzing indexes when you rebuild them. This entire approach depends on the index being analyzed the first time it is built so you have data with which to work.

Conclusion

Last month we talked about how to partition indexes when they become too big. You will see that indexes that are less than a level of three do not usually become bigger than 4M. If you do have indexes larger than 4M, you might also want to make a local 4m tablespace.

Now you can rebuild just the indexes that have either spread out or truly grown, without having to worry about fragmentation in these tablespaces. What a relief!

About the Author

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

Monitoring Index Usage in Oracle9i

B Daniel T. Liu
(View this article, downloadable scripts, and links, online at http://www.dbazine.com/liu3.html)

Introduction

DBAs and developers love indexes. They speed up query searches, especially in a data warehouse environment, where the database receives many ad-hoc requests. To avoid full-table scans, we tend to put indexes on every potentially searchable column. However, Indexes take lot of tablespace storage; in many cases, indexes take more storage space than indexed tables. Indexes also add overhead when inserting and deleting rows. Prior to Oracle9i, it was hard to find out if the index had been used or not used, so many databases have many unused indexes. The purpose of this article is to explain how to identify unused indexes using the new feature in Oracle9i.

Identifying Unused Indexes

Oracle9i provides a new mechanism of monitoring indexes to determine if those indexes are being used or not used. To start monitoring an index's usage, issue this command:

```
ALTER INDEX index_name MONITORING USAGE;
```

To stop monitoring an index, type:

```
ALTER INDEX index_name NOMONITORING USAGE;
```

Oracle contains the index monitoring usage information in the *v$object_usage* view.

```
CREATE OR REPLACE VIEW SYS.V$OBJECT_USAGE
(
    INDEX_NAME,
    TABLE_NAME,
    MONITORING,
    USED,
    START_MONITORING,
    END_MONITORING
)
AS
select io.name, t.name,
       decode(bitand(i.flags, 65536), 0, 'NO', 'YES'),
       decode(bitand(ou.flags, 1), 0, 'NO', 'YES'),
       ou.start_monitoring,
       ou.end_monitoring
from sys.obj$ io, sys.obj$ t, sys.ind$ i, sys.object_usage ou
where io.owner# = userenv('SCHEMAID')
  and i.obj# = ou.obj#
  and io.obj# = ou.obj#
  and t.obj# = i.bo#
/
COMMENT ON TABLE SYS.V$OBJECT_USAGE IS
'Record of index usage'
/
GRANT SELECT ON SYS.V$OBJECT_USAGE TO "PUBLIC"
/
```

The view displays statistics about index usage gathered from the database. Here are the descriptions of the view's columns:

COLUMN NAME	DESCRIPTION
INDEX_NAME:	The index name in sys.obj$.name
TABLE_NAME:	The table name in sys.obj$obj$name
MONITORING:	YES (index is being monitored), NO (index is not being monitored)
USED:	YES (index has been used), NO (index has not been used)
START_MONITORING:	The start monitoring time
END_MONITORING:	the end monitoring time

All indexes that have been used at least once can be monitored and displayed in this view. However, a user can only retrieve its own schema's index usage. Oracle does not provide a view to retrieve all schemas' indexes. To retrieve index usage for all schemas, log in as SYS user and run the following script (Note: this is not an Oracle provided script. The *v$all_object_usage* is a costumed view. It contains one more column, the owner of the index.)

```
$ cat all_object_usage.sql
CREATE OR REPLACE VIEW SYS.V$ALL_OBJECT_USAGE
(
    OWNER,
    INDEX_NAME,
    TABLE_NAME,
    MONITORING,
    USED,
    START_MONITORING,
    END_MONITORING
)
AS
select u.name, io.name, t.name,
       decode(bitand(i.flags, 65536), 0, 'NO', 'YES'),
       decode(bitand(ou.flags, 1), 0, 'NO', 'YES'),
       ou.start_monitoring,
       ou.end_monitoring
from  sys.obj$ io, sys.obj$ t, sys.ind$ i, sys.object_usage ou,
sys.user$ u
where i.obj# = ou.obj#
  and io.obj# = ou.obj#
  and t.obj# = i.bo#
  and io.owner# = u.user#
/
COMMENT ON TABLE SYS.V$ALL_OBJECT_USAGE IS
'Record of all index usage - developed by Daniel Liu'
/
GRANT SELECT ON SYS.V$ALL_OBJECT_USAGE TO "PUBLIC"
/
CREATE PUBLIC SYNONYM V$ALL_OBJECT_USAGE
    FOR SYS.V$ALL_OBJECT_USAGE
/
```

Each time you issue MONITORING USAGE, the view is reset for the specified index. Any previous usage information is cleared or reset, and a new start time is recorded. Every time you issue NOMONITORING USAGE, no further monitoring is performed; the end time is recorded for the monitoring period. If you drop an index that is being monitored, information about that index will be deleted from *v$object_usage* or *v$all_object_usage* view.

Identifying All Unused Indexes in a Database

This script will start monitoring of all indexes:

```
#####################################################################
         ## start_index_monitoring.sh
##

#####################################################################
         #!/bin/ksh
         # input parameter:    1: password
         #                     2: SID
         if (($#<1))
         then
```

```
                 echo "Please enter 'system' user password as the first
parameter !"
                 exit 0
         fi
         if (($#<2))
         then
                 echo "Please enter instance name as the second
parameter!"
                 exit 0
         fi
         sqlplus -s <<!
         system/$1@$2
         set heading off
         set feed off
         set pagesize 200
         set linesize 100
         spool start_index_monitoring.sql
         select 'ALTER INDEX '||OWNER||'.'||INDEX_NAME||' MONITORING
USAGE;'
         from dba_indexes
         where owner not in
('SYS','SYSTEM','OUTLN','AURORA\$JIS\$UTILITY\$');
         spool off
         exit
         !
         sqlplus -s <<!
         oracle/$1@$2
         @./start_index_monitoring.sql
         exit
         !
```

This script will stop monitoring of all indexes:

```
######################################################################
         ## stop_index_monitoring.sh
##

######################################################################
         #!/bin/ksh
         # input parameter:      1: password
         #                       2: SID
         if (($#<1))
         then
                 echo "Please enter 'system' user password as the first
parameter !"
                 exit 0
         fi
         if (($#<2))
         then
                 echo "Please enter instance name as the second
parameter!"
                 exit 0
         fi
         sqlplus -s <<!
         system/$1@$2
         set heading off
         set feed off
         set pagesize 200
         set linesize 100
         spool stop_index_monitoring.sql
         select 'ALTER INDEX '||OWNER||'.'||INDEX_NAME||' NOMONITORING
USAGE;'
         from dba_indexes
         where owner not in
('SYS','SYSTEM','OUTLN','AURORA/$JIS/$UTILITY/$');
         spool off
         exit
         !
         exit
         sqlplus -s <<!
         oracle/$1@$2
         @./stop_index_monitoring.sql
         exit
         !
```

This script will generate a report for all unused indexes:

```
#####################################################################
         ## identify_unused_index.sh
##

#####################################################################
         #!/bin/ksh
         # input parameter:       1: password
         #                        2: SID
         if (($#<1))
         then
                 echo "Please enter 'system' user password as the first
parameter          !"
                 exit 0
         fi
         if (($#<2))
         then
                 echo "Please enter instance name as the second
parameter!"
                 exit 0
         fi
         sqlplus -s <<!
         system/$1@$2
         set feed off
         set pagesize 200
         set linesize 100
         ttitle center "Unused Indexes Report" skip 2
         spool unused_index.rpt
         select owner,index_name,table_name,used
         from v\$all_object_usage
         where used = 'NO';
         spool off
         exit
         !
```

Here is an example of an unused index report:

```
Unused Indexes Report
  OWNER           INDEX_NAME                     TABLE_NAME         USE
  --------------- ------------------------------ -----------------  ---
  HR              DEPT_ID_PK                     DEPARTMENTS        NO
  HR              DEPT_LOCATION_IX               DEPARTMENTS        NO
  HR              EMP_DEPARTMENT_IX              EMPLOYEES          NO
  HR              EMP_EMAIL_UK                   EMPLOYEES          NO
  HR              EMP_EMP_ID_PK                  EMPLOYEES          NO
  HR              EMP_JOB_IX                     EMPLOYEES          NO
  HR              EMP_MANAGER_IX                 EMPLOYEES          NO
  HR              EMP_NAME_IX                    EMPLOYEES          NO
  HR              JHIST_DEPARTMENT_IX            JOB_HISTORY        NO
  HR              JHIST_EMPLOYEE_IX              JOB_HISTORY        NO
  HR              JHIST_EMP_ID_ST_DATE_PK        JOB_HISTORY        NO
  HR              JHIST_JOB_IX                   JOB_HISTORY        NO
  HR              JOB_ID_PK                      JOBS               NO
  HR              LOC_CITY_IX                    LOCATIONS          NO
  HR              LOC_COUNTRY_IX                 LOCATIONS          NO
  HR              LOC_ID_PK                      LOCATIONS          NO
  HR              LOC_STATE_PROVINCE_IX          LOCATIONS          NO
  HR              REG_ID_PK                      REGIONS            NO
  OE              INVENTORY_PK                   INVENTORIES        NO
  OE              INV_PRODUCT_IX                 INVENTORIES        NO
  OE              INV_WAREHOUSE_IX               INVENTORIES        NO
  OE              ITEM_ORDER_IX                  ORDER_ITEMS        NO
  OE              ITEM_PRODUCT_IX                ORDER_ITEMS        NO
  OE              ORDER_ITEMS_PK                 ORDER_ITEMS        NO
  OE              ORDER_ITEMS_UK                 ORDER_ITEMS        NO
  OE              ORDER_PK                       ORDERS             NO
```

Conclusion

Oracle9i provided a new means of monitoring index usage and helps us to identify unused indexes. And the capability to find and drop unused indexes not only helps with insert and delete operations, but also saves storage space. No performance degradation was observed when using index monitoring.

References

Oracle Metalink Support.

Oracle9i Database Administrator's Guide.

"Using Oracle9i Application Server to Build Your Web-Based Database Monitoring Tool," Daniel T. Liu; Select Magazine — November 2001 Volume 8, No. 1.

I would also like to acknowledge the assistance of Husam Tomeh of FARES.

All companies and product names are trademarks or registered trademarks of the respective owners. Please report errors in this article to the author. Neither FARES nor the author warrants that this document is error-free.

About the Author

Daniel Liu is a senior Oracle Database Administrator at First American Real Estate Solutions. He has many years of industry experience in database administration and software development. He has worked with large-scale databases in multi-platform environments. His expertise includes Oracle database administration, performance tuning, Oracle networking and Oracle Application Server. As an Oracle Certified Professional, he also taught Oracle certified DBA classes at Elite Consulting Group in Chicago. Daniel has published article with *SELECT Magazine*. He has also given presentations at IOUG-A Live and Oracle Open World. Daniel can be reached by email at dliu@firstam.com or daniel_t_liu@yahoo.com. Read more of his articles at http://www.dbazine.com.

Oracle Blocksize and Index Tree Structures

By Donald K. Burleson
(View this article, downloadable scripts, and links, online at
http://www.dbazine.com/burleson-b1.html)

Each data block within the Oracle index serves as a “node” in the index tree, with the bottom nodes (leaf blocks) containing pairs of symbolic keys and ROWID values. To properly manage the blocks, Oracle controls the allocation of pointers within each data block. As an Oracle tree grows (by inserting rows into the table), Oracle fills the block, and when full, it splits, creating new index nodes (data blocks) to manage the symbolic keys within the index. Hence, an Oracle index block may contain two types of pointers:

- Pointers to other index nodes (data blocks)
- ROWID pointers to specific table rows

Oracle manages the allocation of pointers within index blocks, and this is the reason why we are unable to specify a *pctused* value (the freelist re-link threshold) for indexes. When we examine an index block structure, we see that the number of entries within each index node is a function of two values:

- The length of the symbolic key
- The blocksize for the index tablespace

Because the blocksize affects the number of keys within each index node, it follows that the blocksize will have an effect on the structure of the index tree. All else being equal, large 32K blocksizes will have more keys, resulting in a flatter index than the same index created in a 2K tablespace. A large blocksize will also reduce the number of consistent gets during index access, improving performance for scattered reads access.

Each data block within the index contains “nodes” in the index tree, with the bottom nodes (leaf blocks) containing pairs of symbolic keys and ROWID values. As an Oracle tree grows (by inserting rows into the table), Oracle fills the block, and when the block is full, it splits, creating new index nodes (data blocks) to manage the symbolic keys within the index. Hence, an Oracle index block may contain pointers to other index nodes or ROWID/Symbolic-key pairs.

The number of entries within each index data block is a function of two values:

- The length of the symbolic key
- The blocksize for the index tablespace

Because the blocksize affects the number of keys within each index block, it follows that the blocksize will have an effect on the structure of the index tree. All else being equal, large 32K blocksizes will have more keys, resulting in a flatter index than the same index created in a 2K tablespace.

According to an article by Christopher Foot (www.dbazine.com/foot3.html): “A bigger block size means more space for key storage in the branch nodes of B-tree indexes, which reduces index height and improves the performance of indexed queries. ”

In any case, there appears to be evidence that block size affects the tree structure, which supports the argument that the size of the data blocks affects the structure of the Oracle index tree.

You can use the large (16K — 32K) blocksize data caches to contain data from indexes or tables that are the object of repeated large scans. Does this really help performance? A small but revealing test can reveal the answer to that question. For the test, the following query will be used against a 9i database that has a database block size of 8K, but also has the 16K cache enabled along with a 16K tablespace:

```
select
      count(*)
from
      scott.hospital
where
      patient_id between 1 and 40000;
```

The SCOTT.HOSPITAL table has 150,000 rows in it and has an index build on the PATIENT_ID column. An EXPLAIN of the query reveals that it uses an index range scan to produce the desired end result:

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
(Cost=41 Card=1 Bytes=4)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (FAST FULL SCAN) OF 'HOSPITAL_PATIENT_ID'
              (NON-UNIQUE) (Cost=41 Card=120002 Bytes=480008)
```

Executing the query (twice, to eliminate parse activity and to cache any data) with the index residing in a standard 8K tablespace produces these runtime statistics:

```
Statistics
---------------------------------------------------
          0  recursive calls
          0  db block gets
        421  consistent gets
          0  physical reads
          0  redo size
        371  bytes sent via SQL*Net to client
        430  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

To test the effectiveness of the new 16K cache and 16K tablespace, the index used by the query will be rebuilt into the 16K tablespace that has the exact same characteristics as the original 8K tablespace, except for the larger blocksize:

```
alter index
      scott.hospital_patient_id
      rebuild nologging noreverse tablespace indx_16k;
```

Once the index is nestled firmly into the 16K tablespace, the query is re-executed (again, twice) with the following runtime statistics being produced:

```
Statistics
---------------------------------------------------
          0  recursive calls
          0  db block gets
        211  consistent gets
          0  physical reads
          0  redo size
        371  bytes sent via SQL*Net to client
        430  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

As you can see, the amount of logical reads has been reduced by half simply by using the new 16K tablespace and accompanying 16K data cache. Clearly, the benefits of properly using the new data caches and multi-block tablespace feature of Oracle9i and later are worth your investigation and trials in your own database.

About the Author

Donald K. Burleson is one of the world's top Oracle database experts. He has written 14 books, published more than 100 articles in national magazines, and serves as editor-in-chief of *Oracle Internals*, a leading Oracle database journal. As a leading corporate database consultant, Don has worked with numerous Fortune 500 corporations creating robust database architectures for mission-critical systems. Don's Web sites are http://www.dba-oracle.com and http://www.remote-dba.net/. Read more of his articles at http://www.dbazine.com.

Partitioning in Oracle9i, Release 2 — Part 1

Learn how to use the various partitioning methods in Oracle9i Release 2.

By Liza Fernandez

(View this article, downloadable scripts, and links, online at http://www.dbazine.com/fernandez2.html)

This is the first part of a two-part article addressing "How To" partition in Oracle9i, Release 2. Part 1 will cover the basics of partitioning and how to partition tables. Part 2 will cover the partitioning of indexes. Part 2 will also draw together the concepts from the entire article into real life examples.

Introduction

Oracle DBAs face an ever growing and demanding work environment. The only thing that may outpace the demands of the work place is the size of the databases themselves.

Database size has grown to a point where they are now measured in the hundreds of gigabytes, and in some cases, several terabytes. The characteristics of very large databases (VLDB) demand a different style of administration. Administering VLDB often includes using the ability to partition tables and indexes.

Since partitioning is such an integral part of VLDB, the remainder of this article will focus on how to partition; specifically, the partitioning of tables in an Oracle9i Release 2 environment. Part 2 of this article will focus on the partitioning of indexes. The complete article will cover:

- Partitioning defined
- When to partition
- Different methods of partitioning

- Partitioning tables
- Partitioning indexes

The organization of this article is modular so you can skip to a specific topic of interest. Each of the table partitioning methods (Range, Hash, List, Range-Hash and Range-List) will have its own section that includes code examples and check scripts.

Background

This article assumes that Oracle9i Release 2 is properly installed and running. You will also need to have a user account that has a minimum of Create Table, Alter Table and Drop Table privileges. In addition to the basic privileges listed above, you will need to create five small tablespaces (TS01, TS02, TS03, TS04, TS05) or changes to the tablespace clause to use the examples provided in this article.

Ideally, you should try each of the scripts in this article under a DBA role. All scripts have been tested on Oracle9i Release 2 (9.2) running on Windows 2000.

Partitioning Defined

The concept of divide and conquer has been around since the times of Sun Tzu (500 B.C.). Recognizing the wisdom of this concept, Oracle applied it to the management of large tables and indexes. Oracle has continued to evolve and refine its partitioning capabilities since its first implementation of range partitioning in Oracle8. In Oracle8i and 9i, Oracle has continued to add both functionality and new partitioning methods. The current version of Oracle9i Release 2 continues this tradition by adding new functionality for list partitioning and the new range-list partitioning method.

When to Partition

There are two main reasons to use partitioning in a VLDB environment. These reasons are related to management and performance improvement. Partitioning offers:

- Management at the individual partition level for data loads, index creation and rebuilding, and backup/recovery. This can result in less down time because only individual partitions being actively managed are unavailable.
- Increased query performance by selecting only from the relevant partitions. This weeding out process eliminates the partitions that do not contain the data needed by the query through a technique called partition pruning.

The decision about exactly when to use partitioning is rather subjective. Some general guidelines that Oracle and I suggest are listed below. Use partitioning:

- When a table reaches a “large” size. Large is defined relative to your environment. Tables greater than 2GB should always be considered for partitioning.
- When performance benefits outweigh the additional management issues related to partitioning.
- When the archiving of data is on a schedule and is repetitive. For instance, data warehouses usually hold data for a specific amount of time (rolling window). Old data is then rolled off to be archived.

Take a moment and evaluate the criteria above to make sure that partitioning is advantageous for your environment. In larger environments partitioning is worth the time to investigate and implement.

Different Methods of Partitioning

Oracle9i, Release 2, has five partitioning methods for tables. They are listed in the table below with a brief description.

PARTITIONING METHOD	BRIEF DESCRIPTION
Range Partitioning	Used when there are logical ranges of data. Possible usage: dates, part numbers, and serial numbers.
Hash Partitioning	Used to spread data evenly over partitions. Possible usage: data has no logical groupings.

PARTITIONING METHOD	BRIEF DESCRIPTION
List Partitioning	Used to list together unrelated data into partitions. Possible usage: a number of states list partitioned into a region.
Composite Range-Hash Partitioning	Used to range partition first, then spreads data into hash partitions. Possible usage: range partition by date of birth then hash partition by name; store the results into the hash partitions.
Composite Range-List Partitioning	Used to range partition first, then spreads data into list artitions. Possible usage: range partition by date of birth then list partition by state, then store the results into the list partitions.

For partitioning indexes, there are global and local indexes. Global indexes provide greater flexibility by allowing indexes to be independent of the partition method used on the table. This allows for the global index to reference different partitions of a single table. Local indexes (while less flexible than global) are easier to manage. Local indexes are mapped to a specific partition. This one-to-one relationship between local index partitions and table partitions allows Oracle the ability to manage local indexes. Partitioning indexes will be the focus of Part 2 of this article.

Detailed examples and code will be provided for each partitioning method in their respective sections. The use of the ENABLE ROW MOVEMENT clause is included in all of the examples of table partitioning to allow row movement if the partition key is updated.

Partitioning Tables

Range Partitioning

Range partitioning was the first partitioning method supported by Oracle in Oracle8. Range partitioning was probably the first partition method because data normally has some sort of logical range. For example, business transactions can be partitioned by various versions of date (start date, transaction date, close date, or date of payment). Range partitioning can also be performed on part numbers, serial numbers, or any other ranges that can be discovered.

The example provided for range partition will be on a table named PARTITION_BY_RANGE (what else would I call it?). The PARTITION_BY_RANGE table holds records that contain the simple personnel data of FIRST_NAME, MIDDLE_INIT, LAST_NAME, BIRTH_MM, BIRTH_DD, and BIRTH_YYYY. The actual partitioning is on the following columns BIRTH_YYYY, BIRTH_MM, and BIRTH_DD. The complete DDL for the PARTITION_BY_RANGE table is provided in the script RANGE_ME.SQL.

A brief explanation of the code follows. Each partition is assigned to its own tablespace. The last partition is the "catch all" partition. By using MAXVALUE, the last partition will contain all the records with values over the second to last partition.

Hash Partitioning

Oracle's hash partitioning distributes data by applying a proprietary hashing algorithm to the partition key and then assigning the data to the appropriate partition. By using hash partitioning, DBAs can partition data that may not have any logical ranges. Also, DBAs do not have to know anything about the actual data itself. Oracle handles all of the distribution of data once the partition key is identified.

The HASH_ME.SQL script is an example of a hash partition table. Please note that the data may not appear to be distributed evenly because of the limited number of inserts applied to the table.

A brief explanation of the code follows. The PARTITION BY HASH line is where the partition key is identified. In this example the partition key is AGE. Once the hashing algorithm is applied each record is distributed to a partition. Each partition is specifically assigned to its own tablespace.

List Partitioning

List partitioning was added as a partitioning method in Oracle9i, Release 1. List partitioning allows for partitions to reflect real-world groupings (e.g., business units and territory regions). List partitioning differs from range partition in that the groupings in list partitioning are not side-by-side or in a logical range. List partitioning gives the DBA the ability to group together seemingly unrelated data into a specific partition.

The LIST_ME.SQL script provides an example of a list partition table. Note the last partition with the DEFAULT value. This DEFAULT value is new in Oracle9i, Release 2.

A brief explanation of the code follows. The PARTITION BY LIST line is where the partition key is identified. In this example, the partition key is STATE. Each partition is explicitly named, contains a specific grouping of VALUES and is contained in its own tablespace. The last partition with the DEFAULT is the "catch all" partition. This catch all partition should be queried periodically to make sure that proper data is being entered.

Composite Range-Hash Partitioning

Composite range-hash partitioning combines both the ease of range partitioning and the benefits of hashing for data placement, striping, and parallelism. Range-hash partitioning is slightly harder to implement. But, with the example provided and a detailed explanation of the code, one can easily learn how to use this powerful partitioning method.

The RANGE_HASH_ME.SQL script provides an example of a composite range-hash partition table.

A brief explanation of the code follows. The PARTITION BY RANGE clause is where we shall begin. The partition key is (BIRTH_YYYY, BIRTH_MM, BIRTH_DD) for the partition. Next, the SUBPARTITION BY HASH clause indicates what the partition key is for the subpartition (in this case FIRST_NAME, MIDDLE_INIT, LAST_NAME). A SUBPARTITION TEMPLATE then defines the subpartition names and their respective tablespace. Subpartitions are automatically named by Oracle by concatenating the partition name, an underscore, and the subpartition name from the template. Remember that the total length of the subpartition name should not be longer than 30 characters including the underscore.

I suggest that, when you actually try to build a range-hash partition table, you do it in the following steps:

1. Determine the partition key for the range.
2. Design a range partition table.
3. Determine the partition key for the hash.
4. Create the SUBPARTITION BY HASH clause.
5. Create the SUBPARTITION TEMPLATE.

Do Steps 1 and 2 first. Then you can insert the code created in Steps 3 — 5 in the range partition table syntax.

Composite Range-List Partitioning

Composite range-list partitioning combines both the ease of range partitioning and the benefits of list partitioning at the subpartition level. Like range-hash partitioning, range-list partitioning needs to be carefully designed. The time used to properly design a range-list partition table pays off during the actual creation of the table.

The RANGE_LIST_ME.SQL script provides an example of a composite range-list partition table.

A brief explanation of the code follows. The PARTITION BY RANGE clause identifies the partition key (BIRTH_YYYY, BIRTH_MM, BIRTH_DD). A SUBPARTITION TEMPLATE then defines the subpartition names and their respective tablespace. Subpartitions are automatically named by Oracle by concatenating the partition name, an underscore, and the subpartition name from the template. Remember that the total length of the subpartition name should not be longer than 30 characters including the underscore.

When building a range-list partition table, you may want to refer to the steps mentioned at the end of the Composite Range-List section. The only difference is in Step 4. Instead of "Create the SUBPARTITION BY HASH clause" it would read, "Create the SUBPARTITION BY LIST clause" for the range-list partition table.

Conclusion

This is the first of a two-part article suggesting the use of partition tables in VLDB environments. Part two of this article will cover partition indexes. In part two both methods (partition tables and indexes) will be brought together in real life examples.

About the Author

Liza Fernandez is an aspiring DBA working toward her Oracle9i DBA certification. She is also pursuing her Master's Degree in Information Systems Management. Read more of her articles at http://www.dbazine.com.

Partitioning in Oracle9i, Release 2 — Part 2

Learn how to use the various partitioning methods in Oracle9i Release 2.

By Liza Fernandez
(View this article, downloadable scripts, and links, online at
http://www.dbazine.com/fernandez3.html)

This is the second part of a two-part article addressing "How To" partition in Oracle9i Release 2. Part 1 covers the basics of partitioning and how to partition tables. Part 2 will cover the partitioning of indexes. Part 2 will also draw together the concepts from the entire article into real-life examples.

Introduction

In Part 1 of "Partitioning in Oracle9i Release 2," we learned how to use the various table partitioning methods in the latest release of Oracle. We will now continue on and learn about Globally Partitioned and Locally Partitioned Indexes. We will cover:

- Background/overview
- Globally partitioned indexes
- Locally partitioned indexes
- When to use which partitioning method
- Real-life example

Background

This article assumes that Oracle9i Release 2 is properly installed and running. You will also need to have a user account that has a minimum of Create Index, Alter Index and Drop Index privileges. In addition to the basic privileges listed above, you will need to create five small tablespaces

(ITS01, ITS02, ITS03, ITS04, ITS05) or changes to the tablespace clause to use the examples provided in this article.

Ideally, you should try each of the scripts in this article under a DBA role. All scripts have been tested on Oracle9i Release 2 (9.2) running on Windows 2000. The examples below build off of the examples that were used in Part 1 of this article.

Globally Partitioned Indexes

There are two types of global indexes, non-partitioned and partitioned. Global non-partitioned indexes are those that are commonly used in OLTP databases (refer to figure1). The syntax for a globally non-partitioned index is the exact same syntax used for a "regular" index on a non-partitioned table. Refer to GNPI_ME.SQL for an example of a global non-partitioned index.

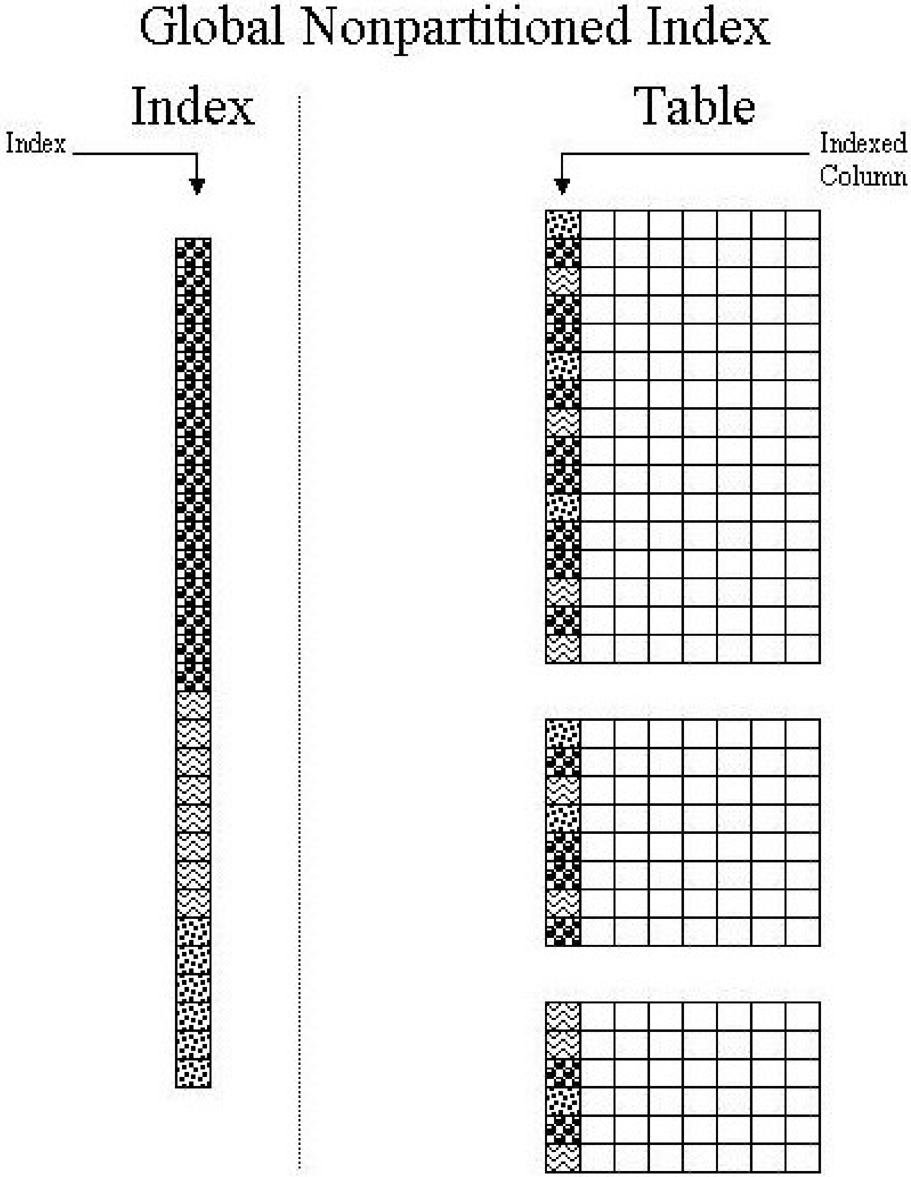


Figure 1 *Global Non-partitioned Index*

The other type of global index is the one that is partitioned. Globally partitioned indexes at this time can only be ranged partitioned and has similar syntactical structure to that of a range-partitioned table. GPI_ME.SQL is provides for an example of a globally partitioned index. Note that a globally partitioned index can be applied to any type of partitioned table. Each partition of the globally partitioned index can and may refer to one or more partitions at the table level. For a visual representation of a global partitioned index refer to figure 2.

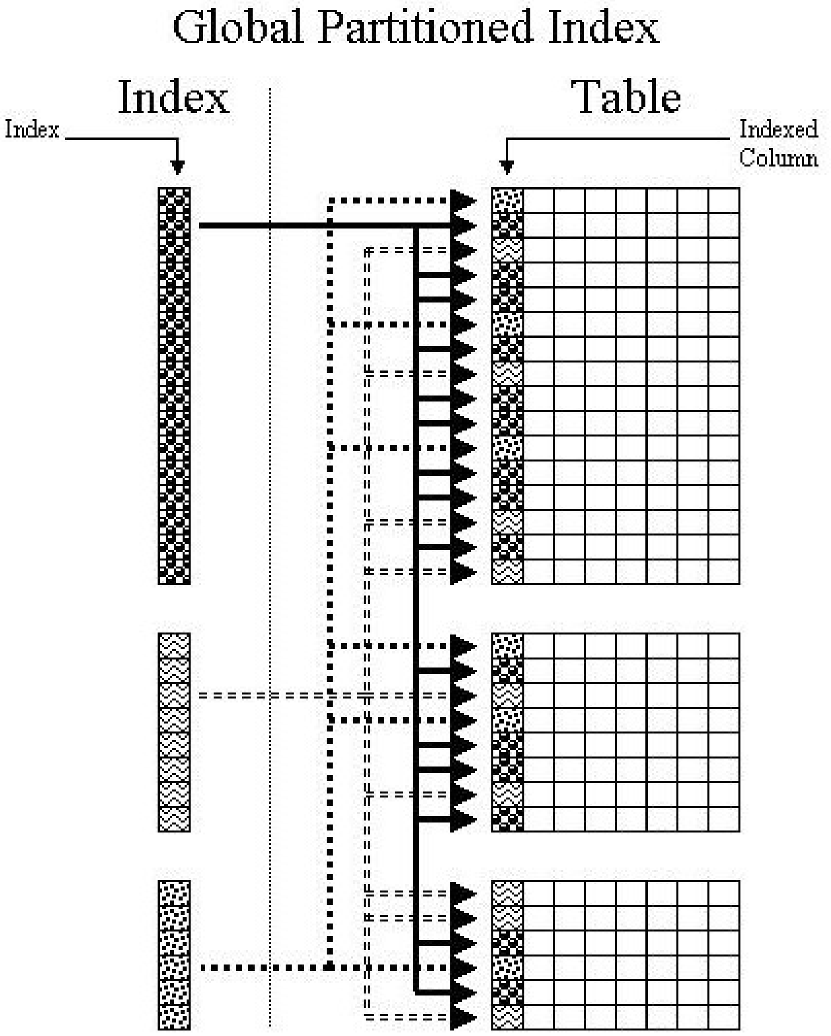


Figure 2 *Global Partitioned Index*

The maintenance on globally partitioned indexes is a little bit more involved compared to the maintenance on locally partitioned indexes. Global indexes need to be rebuilt when there is DDL activity on the underlying table. The reason why they must be rebuilt is that DDL activity often causes the global indexes to be usually marked as UNUSABLE. To correct this problem, there are two options to choose from:

- Use ALTER INDEX <index_name> REBUILD;
- Or use UPDATE GLOBAL INDEX clause when using ALTER TABLE.

The syntax for the ALTER INDEX statement is relatively straightforward, so we will only focus on the UPDATE GLOBAL INDEX clause of the ALTER TABLE statement. The UPDATE GLOBAL INDEX is between the partition specification and the parallel clause. The partition specification can be any of the following:

- ADD PARTITION | SUBPARTITION (hash only)
- COALESCE PARTITION | SUBPARTITION
- DROP PARTITION
- EXCHANGE PARTITION | SUBPARTITION
- MERGE PARTITION
- MOVE PARTITION | SUBPARTITION
- SPLIT PARTITION
- TUNCATE PARTITION | SUBPARTITION

For example:

```
ALTER TABLE <TABLE_NAME>
<PARTITION SPECIFICATION>
UPDATE GLOBAL INDEX
PARALLEL (DEGREE #)
```

Locally Partitioned Indexes

Locally partitioned indexes are for the most part very straightforward. The LPI_ME.SQL script shows examples of this type of index. In the script, locally partitioned indexes are created on three differently partitioned tables (range, hash, and list). Figure 3 gives a visual representation of how a locally partitioned index works.

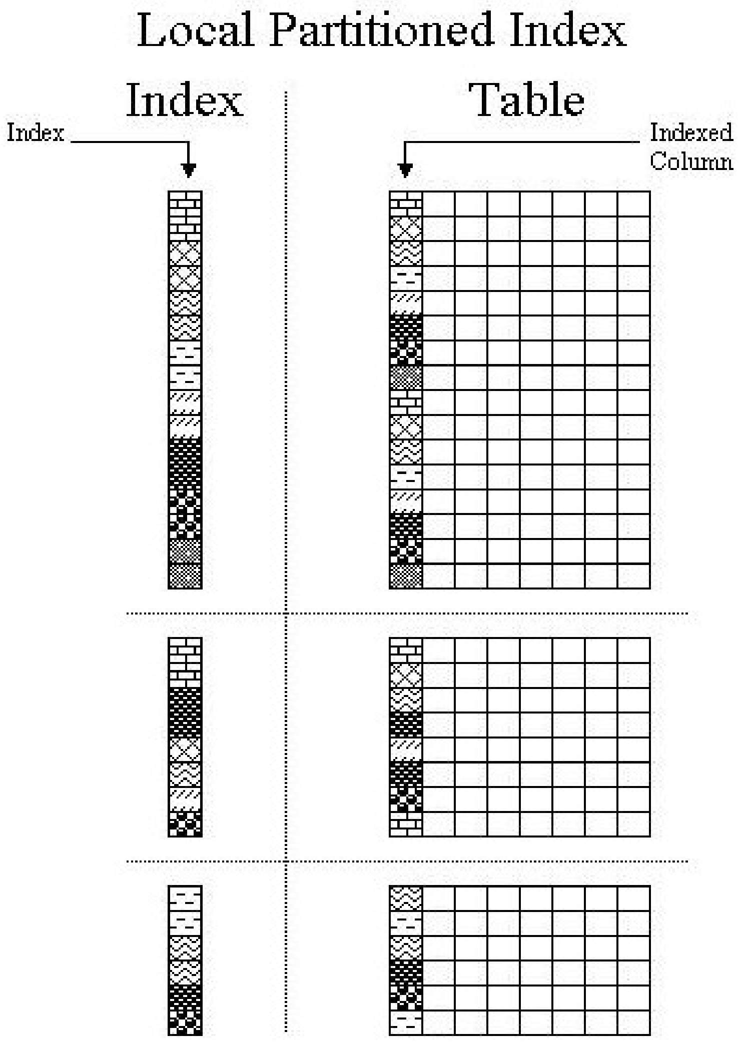


Figure 3 *Local Partitioned Index*

Extra time should be allocated when creating locally partitioned indexes on range-hash or range-list partitioned tables. There are a couple reasons that extra time is needed for this type of index. One of the reasons is that a decision needs to be made on what the index will be referencing in regards to a range-hash or range-list partitioned table. A locally partitioned index can be created to point to either partition level or subpartition level.

Script LPI4CPT1_ME.SQL is the example for the creation of two locally partitioned indexes. This script shows how to create a locally partitioned index on both a range-hash and range-list partitioned table at the partition level. Each of the partitions of the locally partitioned indexes is assigned to its own tablespace for improved performance.

When creating a locally partitioned index, one needs to keep in mind the number of subpartitions of the range-hash or range-list partitioned table being indexed. The reason for this is that the locally partitioned index will need to reference each subpartition of the range-hash or range-list partitioned table. So, for the locally partitioned index created by LPI4CPT2_ME.SQL, this means that one index references 25 different subpartitions. For a visual representation of this, refer to figure 4. Script LPI4CPT3_ME.SQL is provided as an example of locally partitioned index on a range-list partition table.

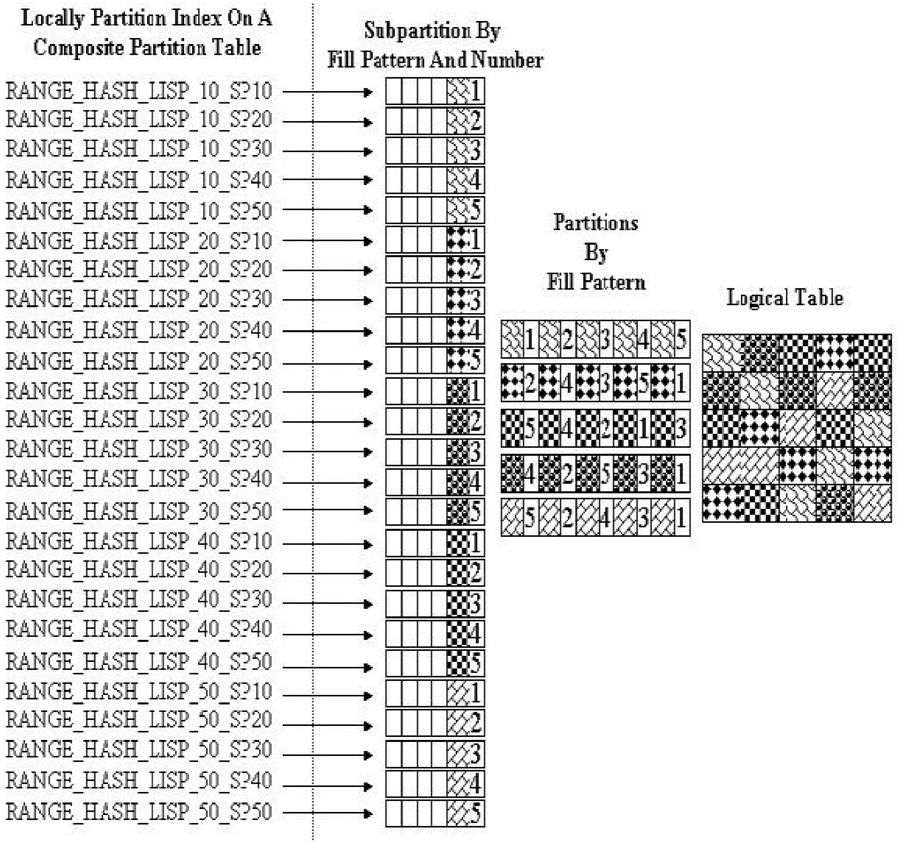


Figure 4 *Locally Partitioned Index*

Note: At this time Oracle has not implemented a SUBPARTITION TEMPLATE clause for the creation of locally partitioned indexes on range-hash or range-list partition tables. This means that you need to type everything out as in the examples in LPI4CPT2_ME.SQL and LPI4CPT3_ME.SQL.

Maintenance of locally partitioned indexes is much easier than the maintenance of globally partitioned indexes. Whenever there is DDL activity on the underlying indexed table, Oracle rebuilds the locally partitioned index.

This automatic rebuilding of locally partitioned indexes is one reason why most DBAs prefer locally partitioned indexes.

When to Use Which Partitioning Method

There are five different table partitioning methods (range, hash, list, range-hash and range-list) and three for indexes (global non-partitioned, global partitioned, and locally partitioned). So, the obvious question is: "When do I use which combination of table and index partitioning?" There is no concrete answer for that question. However, here are some general guidelines on mixing and matching table and index partitioning.

- First, determine whether or not you need to partition the table.
  - Refer to Part 1 of this article under "When To Partition"
- Next, decide which table partitioning method is right for your situation.
  - Each method is described in Part 1 of this article under "Different Methods of Partitioning"
- Determine how volatile the data is.
  - How often are there inserts, updates, and deletes?
- Choose your indexing strategy: global or local partitioned indexes.
  - Each type has its own maintenance consideration.

These guidelines are good place to start when developing a partitioning solution.

Real-life Example

The "rolling window" concept of only retaining a certain amount of data is the norm in most data warehousing environments. This rolling window can also used to archive data from an OLTP system. For our example, we will assume that there is a twelve month rolling window.
Our example will cover the following steps:

- Create a range partition table that has a locally partitioned index.
- Use "CREATE TABLE . . AS" to copy the data into a separate table.

- Archive off the table created to hold the rolled off data.
- Drop last month partition.
- Add new months partition.

Script EXAMPLE.sql is an annotated code of the example above.

Conclusion

During the course of this two-part article, we have covered the "How to" of partitioning in Oracle9i Release 2. Part 1 covered the basics of table partitioning. Part 2 followed with partitioning of indexes. We then brought together both partitioning methods and evaluated when to use each method. Near the end of this article, we applied what we have learned in a real-life example. I hope that this article gives you the basic knowledge to evaluate and use partitioning in your next design and implementation of Oracle9i Release 2.

About the Author

Liza Fernandez is an aspiring DBA working toward her Oracle9i DBA certification. She is also pursuing her Master's Degree in Information Systems Management. Read more of her articles at http://www.dbazine.com.

Understanding Bitmap Indexes

By Jonathan Lewis
(View this article, downloadable scripts, and links, online at http://www.dbazine.com/jlewis3.html)

Bitmap indexes are a great boon to certain kinds of application, but there is a lot of misinformation in the field about how they work, when to use them, and the side-effects. This article examines the structure of bitmap indexes, and tries to explain how some of the more commonly repeated misconceptions came into existence.

Everybody Knows ...

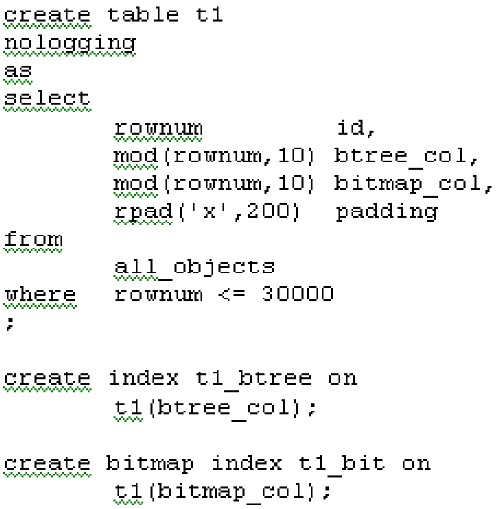
If you did a quick survey to discover the understanding that people had of bitmap indexes, you would probably find the following comments being quoted fairly frequently:

- When there are bitmap indexes on tables, then updates will take out full table locks.
- Bitmap indexes are good for low-cardinality columns.
- Bitmap index scans are more efficient than tablescans even when returning a large fraction of a table.
- The third claim is really little more than a (possibly untested) corollary to the second claim. And all three claims are in that grey area somewhere between false and extremely misleading.
- Of course, there is a faint element of truth to these claims — just enough to explain why they should have arisen in the first place.
- The purpose of this article is to examine the structure of bitmap indexes, review the claims, and try to sort out some of the costs and benefits of using bitmap indexes.

What Is a Bitmap Index?

Indexes are created to allow Oracle to identify requested rows as efficiently as possible. Bitmap indexes are no exception — however, the strategy behind bitmap indexes is very different from the strategy behind B*tree indexes. To demonstrate this, we can start by examining a few block dumps.

Consider the SQL script in figure 1.

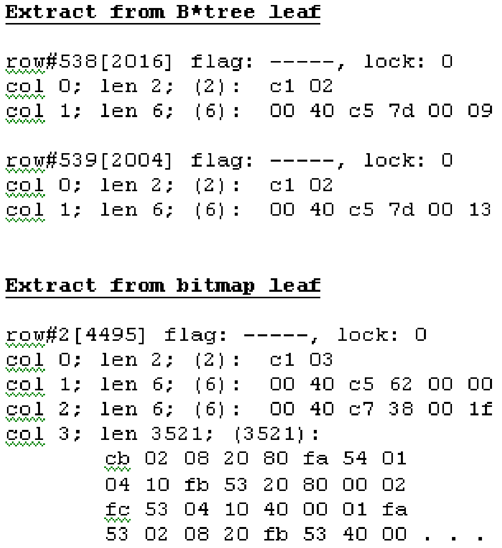


```
create table t1
nologging
as
select
        rownum          id,
        mod(rownum,10)  btree_col,
        mod(rownum,10)  bitmap_col,
        rpad('x',200)   padding
from
        all_objects
where   rownum <= 30000
;

create index t1_btree on
        t1(btree_col);

create bitmap index t1_bit on
        t1(bitmap_col);
```

Figure 1: *Sample data.*

Note how we have defined the *btree_col* and *bitmap_col* so that they hold identical data that cycles through the values zero to nine.

On a 9.2 database with a block size of 8K, the resulting table was 882 blocks long. The B*tree index had 57 leaf blocks, and the bitmap index had 10 leaf blocks.



```
Extract from B*tree leaf

row#538[2016] flag: -----, lock: 0
col 0; len 2; (2):  c1 02
col 1; len 6; (6):  00 40 c5 7d 00 09

row#539[2004] flag: -----, lock: 0
col 0; len 2; (2):  c1 02
col 1; len 6; (6):  00 40 c5 7d 00 13


Extract from bitmap leaf

row#2[4495] flag: -----, lock: 0
col 0; len 2; (2):  c1 03
col 1; len 6; (6):  00 40 c5 62 00 00
col 2; len 6; (6):  00 40 c7 38 00 1f
col 3; len 3521; (3521):
       cb 02 08 20 80 fa 54 01
       04 10 fb 53 20 80 00 02
       fc 53 04 10 40 00 01 fa
       53 02 08 20 fb 53 40 00 . . .
```

Figure 2: *Symbolic block dumps.*

Clearly the bitmap index was in some way much more tightly packed than the B*tree index. To see the packing, we can produce a symbolic dump from the indexes using commands like:

```
alter system
dump datafile x block y;
```

See figure 2 for results — be warned, however, that symbolic block dumps can be a little misleading. Some of the information they display is derived, some is re-arranged for the sake of clarity.

Do Bitmaps Lock Tables?

Looking at figure 2, we see in the B*tree index that every entry consists of a set of flags, a lock byte, and (in this case) two columns of data. The two columns are in fact the indexed value, and a rowid — and every row in our table has a corresponding entry of this form in the index. (If the index were a unique index, we would still see the same content in each entry, but the layout would be a little different).

In the bitmap index, every entry consists of a set of flags, a lock byte, and (in this case) four columns of data. The four columns are in fact the indexed value, a pair of rowids and a stream of bits. The pair of rowids identifies a contiguous section of the table, and the stream of bits is encoded to tell us which rows in that range of rowids hold that value.

Look at the size of the bit stream though — the length of the column in the example above is 3,521 bytes, or roughly 27,000 bits. Allowing about 12 percent overhead for check sums and so on, this single entry could cover about 24,000 rows in the table. But there is only one lock byte for the entire entry, which means a single lock will have some sort of impact on as many as 24,000 rows in the table.

So this is where that dubious claim originates — if you think that a bitmap index causes a full table lock, then you have been experimenting with tables that are too small.

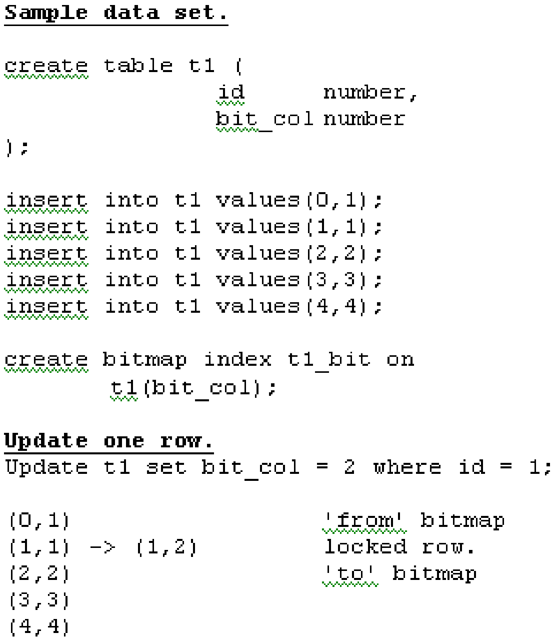
A single bitmap lock could cover thousands of rows — which is pretty bad news — but it does not lock the table.

Consequences of Bitmap Locks

We shouldn't stop with that conclusion, though, as it would be easy to misinterpret the result. We need to understand what actions will cause that one critical lock byte to be taken, and exactly what effect that will have on the thousands of related rows.

We can investigate this with a much smaller test (see figure 3). We start by building a small table, and then doing different updates to different rows in that table.

Sample data set:



```
Sample data set.

create table t1 (
                id       number,
                bit_col number
);

insert into t1 values(0,1);
insert into t1 values(1,1);
insert into t1 values(2,2);
insert into t1 values(3,3);
insert into t1 values(4,4);

create bitmap index t1_bit on
       t1(bit_col);

Update one row.
Update t1 set bit_col = 2 where id = 1;

(0,1)                       'from' bitmap
(1,1) -> (1,2)              locked row.
(2,2)                       'to' bitmap
(3,3)
(4,4)
```

Figure 3: *Preparing for update tests.*

Note that we have updated the indexed column of one row in the table. If we dump the index and table blocks, we will see that there is a lock byte set on that one row in the table, but two sections of the bitmap index are locked. The two sections will be the section for nearby rows where the current value is 1 (the “from” section) and the section for nearby rows where the value is 2 (the “to” section). (In fact we should see that those two sections of the bitmap have been copied and both copies are locked).

The question we have to pursue now, is how aggressive is Oracle’s locking in this case.

The answer may come as a bit of a surprise to those who think in terms of “bitmap indexes cause table locks.”

We can do any of the following (each one is a separate test).

Update a row in the “from” section, provided we do not try to update the bitmap column.

```
update t1
set id = 5
where id = 0;
```

Update a row in the “to” section, provided we do not try to update the bitmap column.

```
update t1
set id = 6
where bit_col = 2;
```

These tests show us that a row can be covered by a locked bitmap section, and still be available for update.

Lock collisions are possible, of course, for example neither of the following statements is updating a locked table row, but either of them would cause their session to wait on a “TX” lock in mode 4 (shared):

```
update t1
set bit_col = 4
where id = 2; – bit_col = 2
update t1
set bit_col = 2
where id = 3 – bit_col = 3
```

Note, however, that the problem requires two things to be true. First, we must be updating the indexed column, and secondly, the row that we are updating must be covered by a previously locked bitmap section, i.e., it must be “fairly near” another row that is in mid-update, and there is a strictly limited list of values (*viz*: 4 values) that could cause a collision.

Bear in mind that we can, with our sample scenario, update the bitmap indexed column in a nearby row, provided that neither the initial nor final value is 1 or 2. For example:

```
update t1
set bit_col = 4
where bit_col = 3;
```

So, bitmap indexes do NOT cause table locks; and if our updates do not affect the bitmapped column, the presence of the bitmap indexes causes no problems at all, and even if our updates do update bitmapped columns, we may be able to engineer a set of non-colliding updates.

Problems with Bitmaps

Of course, there are some problems with using bitmaps that go beyond the question of update collisions.

Remember that inserts and deletes on a table will result in updates to all the associated indexes. Given the large number of rows covered by a single bitmap index entry, any degree of concurrency of inserts or deletes has a fairly high chance of affecting overlapping index sections and causing massive contention.

Moreover, even serialized DML that affects bitmap indexes may have a more significant performance impact than you would expect.

I pointed out that a simple update to a single row typically results in an entire bitmap section being copied. Look back at (figure 1), and remind yourself how big a single bitmap section could be. In the example it was 3,500 bytes, (in Oracle9 the limit is close to half a block). You can find that a small number of changes to your data can have a surprisingly large impact on the size of any bitmap index that gets updated as a consequence.

You can get lucky — but in general, you should start with the assumption that even a serialized batch update will be most effective if you drop the bitmap indexes before the batch and rebuild them afterwards.

Low Cardinality Columns

It is often claimed, "bitmap indexes are good for low cardinality columns." If we are a little fussy about the language, we might prefer to say "low distinct cardinality." In either case, the intent is to identify columns that hold a relatively small number of different values.

This is indeed a reasonably accurate statement — provided it is qualified and explained properly. Unfortunately, many people seem to think that this means a bitmap index is magically so efficient that you can use it to access large fractions of a table in a way that would not be considered sensible with a B*tree index.

The classic example quoted for bitmap indexing is the extreme one of sex; a column holding just two values (or three if you include the "n/a" dictated by the ISO standard). We will be slightly less extreme, and consider an example based on the countries that make up the United Kingdom — England, Ireland, Scotland, and Wales.

Assume we have a block size of 8K, and a (reasonably ordinary) row size of 200 bytes, for a total of 40 rows per block. Insert a few million rows into that table, ensuring that the distribution of the four countries is uniformly random. There will be roughly ten rows per block for each country.

If I use the bitmap index to access all the rows for England, I will visit every block in the table (ten times) in order. Surely it would be more efficient to do a tablescan than to use that index.

In fact, even if I expand my data set to 40 countries, I am still likely to find one row in each block in the table. Perhaps by the time my data has expanded to global proportions (say 640 countries so that any given country appears once every 16 blocks), it might be cheaper to access the data by index rather than by tablescan. But a column with 640 different values hardly seems to qualify, at first sight, for the description of "low distinct cardinality."

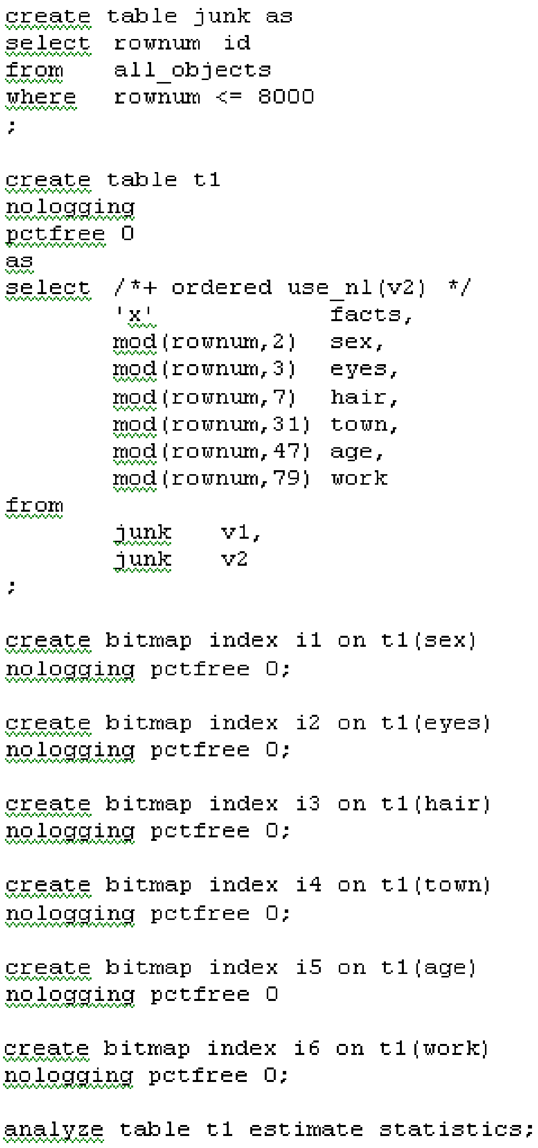
Of course, descriptive expressions like "low," "small," "close to zero" need some qualification. Is 10,000 close to zero, for example? If the alternative is ten billion, then the answer is yes!

Forget the vague expressions like "low cardinality." In most cases there are only two points to bear in mind when considering bitmap indexes. First, it is the number of different blocks in the table that you have to visit for a typical index value that is the major cost of using an individual index; changing an index from B*tree to bitmap won't magically make it a better index. Secondly, it is Oracle's optimizer mechanism for combining multiple bitmap indexes that makes them useful.

Consider this example based on the UK population of roughly 64M people:

- 50M have brown eyes
- 35M are female
- 17M have black hair
- 1.8M live in the Birmingham area
- 1.2M are aged 25
- 750,000 work in London

Any one fact gives us a huge number of people — but how many brown-eyed, black-haired, women aged 25 live in Birmingham and work in London? Perhaps a couple of dozen.



```
create table junk as
select  rownum  id
from    all_objects
where   rownum <= 8000
;

create table t1
nologging
pctfree 0
as
select  /*+ ordered use_nl(v2) */
        'x'             facts,
        mod(rownum,2)   sex,
        mod(rownum,3)   eyes,
        mod(rownum,7)   hair,
        mod(rownum,31)  town,
        mod(rownum,47)  age,
        mod(rownum,79)  work
from
        junk    v1,
        junk    v2
;

create bitmap index i1 on t1(sex)
nologging pctfree 0;

create bitmap index i2 on t1(eyes)
nologging pctfree 0;

create bitmap index i3 on t1(hair)
nologging pctfree 0;

create bitmap index i4 on t1(town)
nologging pctfree 0;

create bitmap index i5 on t1(age)
nologging pctfree 0

create bitmap index i6 on t1(work)
nologging pctfree 0;

analyze table t1 estimate statistics;
```

Figure 4: *Modeling the UK population.*

Individually an index (B*tree or bitmap) on any one of these facts would be completely useless if we translated this data set and query into an Oracle database.

A multi-column B*tree index on all six facts might be quite helpful — until we decided to ask for men who were six foot tall with beards, instead of women with brown eyes and black hair. You might like to try the experiment (see figure 4), which needs about 2.0GB of free space and may take a couple of hours to complete at around the 500MHz CPU mark.

Due to restrictions on space, I built a smaller model — emulating a population of only 36 million. The time to build, and size of objects came out as follows on a 600MHz, Win2000 box running 9.2.0.1.

OBJECT	SIZE (MB)	BUILD TIME (MI:SS)
T1	845	16:12
I1 (sex)	11	1:39
I2 (eyes)	16	1:43
I2 (hair)	37	2:17
I4 (town)	40	2:25
I5 (age)	42	2:28
I6 (work)	45	2:42

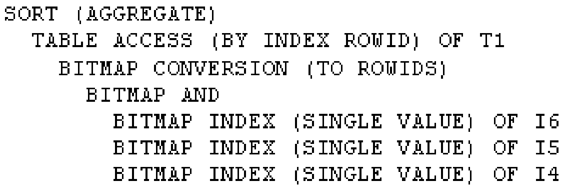
Note particularly the total space taken by the indexes — 191 MB. Just one multi-column index on the same six columns (even with maximum compression) would take at least 430MB, using I don't know how much CPU time to build; and not many systems would have catered for a full in-memory build as the required *sort_area_size* would be about 900MB. So what can all these bitmap indexes do for us? Consider the query:

```
select  count(facts)
from    t1
where   eyes = 1
and     sex = 1
and     hair = 1
and     town = 15
and     age = 25
and     work = 40
;
```

On the reduced data set that I had created, a hint to use a full tablescan, resulted in a run time of one minute and 20 seconds (returning the answer

eight). Of course, with a real set of fact data, the table would have been much bigger, and the time much greater.
With a full, six-column, 430MB index, this query would probably have returned in the time it takes to do about ten physical reads (one table block for each row, and a couple extra for reading index blocks) — the proverbial sub-second response.

With the bitmap indexes as defined, the response time was five seconds. Most of that time was spent in bitmap index range-scans that did physical reads to get index blocks into memory. The actual execution path is shown in figure 5.



```
SORT (AGGREGATE)
  TABLE ACCESS (BY INDEX ROWID) OF T1
    BITMAP CONVERSION (TO ROWIDS)
      BITMAP AND
        BITMAP INDEX (SINGLE VALUE) OF I6
        BITMAP INDEX (SINGLE VALUE) OF I5
        BITMAP INDEX (SINGLE VALUE) OF I4
```

Figure 5: *Execution path.*

There are two interesting points to consider with this result. First, Oracle ignored the three "worst" (i.e., least selective) indexes. Second, although the response time seems slow, the index sizes are so small that it is feasible to think about keeping them in a large *buffer_pool_keep* (or, for Oracle9, *db_keep_cache_size*) to eliminate the cost of the physical reads — an option that would probably not be feasible if you needed several multi-column B*tree indexes to do the same job.

Let's think about the ignored indexes — it is possible for a bitmap plan like this to use an apparently arbitrary number of indexes, and I have seen cases in which Oracle has used more indexes than the limit of five that applies to the *and_equal* access path for single-column B*tree indexes.

The three missing indexes have not been ignored because of some artificial limit. The cost-based optimizer weighs the cost of reading each extra index against the additional precision gained, so bitmap indexes on the classic (male, female) column tend to be ignored despite claims to the contrary. (Delete the clause "work = 40" from the sample query, though, and you will see the index on column "sex" is actually used).

Of course, such bitmaps can be built very quickly and tend to be very small, so you might want to build them anyway, just in case.

Sizing

The sizing of indexes, and the option for maximum buffering have to be considered in the cost, of course, and the question often arises — how big will a bitmap index be?

In the example above, I have tried to build a worst-case scenario, making it as hard as possible for Oracle to gain any advantages in compression.

In the worst case, the size of a bitmap in bits would be:

```
Number of different possible values for the column *
Number of rows that Oracle thinks could fit in a block *
Number of blocks below high water mark.
```

Add about ten percent for checksum information and overhead, and divide by eight to get the number of bytes.

Fortunately, Oracle has some steps for reducing the size of the wasted space — the most important of which is the command to tell Oracle exactly how many rows per block you have in the worst case in a specific table:

```
Alter table XXX
      minimize records_per_block;
```

However, apart from keeping Oracle informed with this command, you also find that the size of the index is very strongly affected by the clustering of the data.

In my example, I have constructed the data to be as thinly scattered as possible — for example, the town column rotates through the values 0 to 30. If I restructure (effectively sort) the data so that all the towns with code 0 are together, followed by all the towns with code 1, the size of the index drops from 40MB to a mere 7MB.

This dramatic variation in size is yet another reason for reviewing the claim about "low cardinality." The potential benefit of a bitmap index varies with the clustering of data (as does the potential benefit of a B*tree index, of course). When you are considering bitmap indexes, do not be put off by a column, which has a "large" number of different values. If every value appears a "large" number of times, and if the rows for each value are reasonably clustered, then a bitmap index may be highly appropriate. In a table with 100M rows, a column with 10,000 different values could easily turn out to be a perfect candidate for a bitmap index.

Conclusion

There are several seriously misleading statements commonly made about bitmap indexes. Some may lead you into avoiding bitmap indexes when they could be very useful, others may lead you into creating bitmap indexes that are totally inappropriate.

Fortunately it is quite hard to make big mistakes with bitmap indexes, but it is a good idea to have some idea of what Oracle does with them so that you can make best use of them.

The key facts to remember are:

- If a B*tree index is not an efficient mechanism for accessing data, it is unlikely to become more efficient simply because you convert it to a bitmap index.
- Bitmap indexes can usually be built quickly, and tend to be surprisingly small.
- The size of the bitmap index varies dramatically with the distribution of the data.
- Bitmap indexes are typically useful only for queries that can use several such indexes at once.

- Updates to bitmapped columns, and general insertion/deletion of data can cause serious lock contention.
- Updates to bitmapped columns, and general insertion/deletion of data can degrade the quality of the indexes quite dramatically.

Remember, too, that the optimizer improves with every release of Oracle. The boundary between the utilization mechanisms for B*tree and bitmap indexes becomes increasingly blurred with the evolution of compressed indexes, index skip scans, and B*tree to bitmap conversions.

References

Oracle9i Release 2 Data Warehousing Guide, Chapter 6.

About the Author

Jonathan Lewis is a freelance consultant with more than 17 years experience in Oracle. He specializes in physical database design and the strategic use of the Oracle database engine, is author of *Practical Oracle8i - Building Efficient Databases* published by Addison-Wesley, and is one of the best-known speakers on the UK Oracle circuit. Further details of his published papers, tutorials, and seminars can be found at http://www.jlcomp.demon.co.uk, which also hosts *The Co-operative Oracle Users' FAQ* for the Oracle-related Usenet newsgroups. Read more of his articles at http://www.dbazine.com.