module18-tuning.htm; updated June18, 2012

**Module 18 – Database Tuning   
(and Other Performance Issues)**

**Objectives**

* Identify how poor system performance results from a poor database system design.
* Familiarize with how database tuning focuses on identifying and fixing underlying flaws.
* Familiarize with typical objects that can be tuned.

**TUNING APPLICATION DESIGN**

**Effective Table Design**

* Poor table design always leads to poor performance.
* Rigid adherence to **fully normalized** relational table guidelines can also result in poor physical performance.  These inefficiencies result from:
  + requiring too many **joins**.
  + failure to reflect the **normal access paths** for data.
* Queries with **large numbers of columns** that come from multiple tables can cause performance to suffer because several tables must be joined.
* Design options include:
  + **Denormalizing** 1NF, 2NF and 3NF solutions.
  + Creating small **summary tables** from large, static tables that stores data in the format in which users ask for the data - this avoids joins where data are often requested and the data doesn't change very often.
  + Separating individual tables into several tables by either **vertical partitioning** and/or **horizontal partitioning**.

**Distribution of CPU Requirements**

* An Oracle database that is **CPU-Bound** (limited by CPU resources) as opposed to **Wait-Bound** (waiting on disk writes of some type) is one that's effectively designed.  This means that other resources are not limiting the database.
* Schedule **long-running batch query/update programs** for off-peak hours; then run them at normal priority.
* Store data in its most appropriate place in a **distributed** **computing** environment - this distributes computing CPU requirements from one server to another.
* Use the **Parallel Query** option to distribute processing requirements of selected SQL statements among multiple CPUs if they are available.

**Effective Application Design**

* There are two principles to follow here:

**1.** limit the number of times that users access the database, and

**2.** coordinate the requests of users for data.  This requires you to know **how** users tend to access data.

* Try to use the **same queries** to handle similar application data requirements; this will increase the likelihood that a data requirement can be resolved by data already in the **SGA**.
* Use **snapshots**, which are **non-updatable views** of part of a database that can be distributed to support typical managerial querying.
* Create stored procedures, functions, and packages and compile them to **eliminate run-time compilation**. The parsed version may exist in the **Shared SQL Pool**.

**1.** The **SQL Text** for all procedures, functions, and packages can be viewed in the **TEXT** column of the **DBA\_SOURCE** view.

**2.** These objects (procedural code) are stored in the **SYSTEM** tablespace so you need to allocate more space to it -- usually double its size.

**TUNING SQL**

* Most **SQL tuning** requires the DBA to work with an application developer.
* Most improvement in database processing will come from tuning SQL.
* The key to SQL tuning is to **minimize the search path** that a database uses to find data.  For the most part this requires the creation of appropriate **indexes** that the database engine will use to find data.

**Indexes**

* An **Index** enables Oracle to locate a row according to its physical location in a datafile by going to the correct file, finding the correct block, then finding the correct row within the block.  Taken together, these values are like **relative record numbers** in a relative file.
* The **File ID** portion of **ROWID** can be compared to the **FILE\_ID** column of the **DBA\_DATA\_FILES** view to determine the file to which an index belongs.
* A query with no **WHERE** clause normally results in a full table scan, reading every block in a table.
* A query for specific rows may cause use of an **index**.
  + The index maps logical values in a table (key columns) to their **ROWIDs** which enables location of the rows directly by their physical location.
* You may index several columns together to form a **concatenated index**.
  + A concatenated index is only used if its **leading column** is used in the query's **WHERE** clause.
  + Consider the following example:

**CREATE INDEX City\_state\_zip\_ndx**   
**ON Employee (City, State, Zip)**   
**TABLESPACE Indexes;**

**SELECT \***   
**FROM Employee**   
**WHERE State = 'NJ';**

* The index is **NOT** used because the **WHERE** clause value of **State** does not match the leading column (**City**) of the index.
* This example would only add overhead because the index would be maintained, even if it's not used.
* The index should be recreated with proper ordering of the component fields in the index if the query is executed often.

**Ordering Data**

* While **row ordering** does not matter in relational theory, it is important to order rows as much as possible when tables are initially created, e.g. when you are porting a system into Oracle from another platform or DBMS.
* Ordered rows may enable Oracle to find needed rows while minimizing the data blocks that are retrieved where users execute queries that specify **ranges of values** (recall the **BETWEEN** operator in SQL for range of value queries).
* Consider the following example which will require fewer data blocks to be read if the records are physically ordered on the **Empno** field.

**SELECT \***   
**FROM Employee**   
**WHERE Empno BETWEEN 1 and 100;**

        You can physically **sort** table rows by SELECTing them to another file with use of the ORDER BY clause, then truncating the original table and loading the rows back into the original table.

**Clusters**

* We covered indexed clusters in an earlier module.
* Another type of cluster, the ***hash cluster***, stores rows in a specific location based on its value in the cluster key column.
  + Every time a row is inserted, its **cluster key value** is used to determine which block to store the row in.
  + This enables hashing directly to data blocks without use of an index.
  + The hash cluster is only used with **equivalence** queries - where the exact value stored in a column is to be found.

**Explain Plan**

* The **EXPLAIN PLAN** command shows the execution path for a query and stores this information to a table (**PLAN\_TABLE**) in the database. You can then query the table. Example:

**EXPLAIN PLAN**   
**SET Statement\_id = 'TEST'**   
**FOR**

**SELECT \***   
**FROM Employees**   
**WHERE last\_name > 'Y%';**

* The query above is not actually executed; rather the plan for execution is stored to the **PLAN\_TABLE**.
* Your account must have a **PLAN\_TABLE** in your schema.  The script to create this table is **UTLXPLAN.SQL** and is located in the **$ORACLE\_HOME/rdbms/admin** subdirectory.
* Query the table to produce the output that shows the execution path.

**SELECT LPAD(' ',2\*LEVEL) || operation**   
**|| ' ' || options ||**   
**' ' || object\_name Path\_Plan**   
**FROM Plan\_Table**   
**WHERE Statement\_id = 'TEST'**   
**CONNECT BY PRIOR Id = Parent\_id**   
**AND Statement\_id = 'TEST'**

**START WITH Id=1;**

**Path\_Plan**   
**-----------------------------------**   
**TABLE ACCESS BY ROWID EMPLOYEE**   
**INDEX RANGE SCAN CITY\_ST\_ZIP\_NDX**

* The output shows that data will be accessed by ROWID through an index range scan of the named index.
* Alternatively, you can also use the **SET AUTOTRACE ON** command in **SQL\*Plus** to generate the explain plan output and trace information for every query that you run.
* Evaluate the output by ensuring that the most selective (most nearly unique) indexes are used by a query.

**TUNING MEMORY USAGE**

You can use the Oracle Enterprise Manager software to analyze usage of memory by Oracle’s various memory caches.

* The dictionary cache in memory is not directly sized or tuned as it is part of the **Shared SQL Pool**.
* These memory areas are managed by the **LRU** (least recently used) algorithm.  You set the **Shared SQL Pool size** with the **SHARED\_POOL\_SIZE** parameter.
* If your **Shared SQL Pool** is too large, then you are wasting memory.
* The **Hit Ratio** measures how well the data buffer cache handles requests.

**Hit Ratio = (Logical Reads - Physical Reads) /**   
**Logical Reads**

* A **perfect ratio is 1.00** - all reads are logical reads; of course, this is generally impossible to obtain since it indicates that all the data that a system user will ever need to access is stored in the **SGA**.
* On-line transaction processing applications should have Hit Ratios in excess of **0.90**.
* If processing for the **Hit Ratio** is within tolerance, you need to check to see if you can reduce the size of the Shared SQL Pool and still maintain a good Hit Ratio.
* Add the following to the **INIT.ORA** file.

**DB\_BLOCK\_LRU\_STATISTICS = TRUE**

* Shutdown the database and restart it.
* The system dictionary table **SYS.X$KCBCBH** maintains memory statistics.  One row is maintained for each buffer in the buffer cache.  You can query this information to determine how many buffers are not being used.
* Use the following query to determine how many cache hits (the **COUNT** column) would be lost if you reduced the number of buffers (the **INDX** column).

**SELECT Sum(Count) Lost\_Hits**   
**FROM Sys.X$Kcbcbh**   
**WHERE indx >= *New\_Number\_Of\_Buffers*;**

**(NOTE: You supply the value in the WHERE clause)**

* If you have **lost hits**, the system will require additional physical reads - the Hit Ratio for this new number of data buffers is:

**Hit Ratio =**   
**(Logical Reads - Physical Reads - Lost Hits)**   
**/ Logical Reads**

* Since running the database in a statistics gathering mode will slow it down due to the additional overhead, you should comment out the **DB\_BLOCK\_LRU\_STATISTICS** parameter after you have finished tuning and restart the database.

**TUNING DATA STORAGE**

**Defragmentation of Segments**

**Fragmented** tables with **multiple extents** will slow down query processing.  This can also slow down the storage of new records because the database may have to dynamically combine free extents to create a new extent large enough to meet the storage parameters of the object where data are being stored.

* We know that a **segment** is created to hold data associated with a new object (index or table) when an object is created.
* The space allocated is used unless the **segment** is released (dropped) or truncated (tables only).
* It would be best if each segment was composed of a single large **INITIAL** extent - compute the size for the initial extent such that it is large enough to handle all of a segment's data.
* Use the **DBA\_SEGMENTS** data dictionary view to determine which segments are comprised of ten or more extents.

**SELECT Tablespace\_name TSName, Owner,**   
**Segment\_Name SNName,**   
**Segment\_type SNType, Extents,**

**Blocks, Bytes**   
**FROM Sys.DBA\_Segments;**

**TSNAME OWNER SNNAME    SNTYPE EXTENTS BLOCKS BYTES**   
**DATA   DBOCK LONGTIME  TABLE  1       15     61440**   
**DATA   DBOCK MAGAZINE  TABLE  1       15     61440**   
**DATA   DBOCK MATH      TABLE  1       15     61440**   
**DATA   DBOCK WORKERANDSKILL CLUSTER 2 30    122880**

* To see the size of the individual extents for a segment, query the **DBA\_EXTENTS** view.  Supply the type of segment you desire (TABLE, INDEX, CLUSTER, ROLLBACK, TEMPORARY, etc.).

**SELECT Tablespace\_name TSNAME, Owner,**   
**Segment\_Name SNNAME,**   
**Segment\_Type SNTYPE, Extent\_id EID,**   
**File\_id FID, Block\_id BID, Bytes, Blocks**   
**FROM Sys.DBA\_Extents**   
**WHERE Segment\_type = '*segment\_name*'**   
**ORDER BY Extent\_id;**

**TSNAME OWNER SNNAME     SNTYPE EID FID BID Bytes Blocks**  
**DATA   DBOCK SYS\_C00890 INDEX  0   4   137 61440 15**   
**DATA   DBOCK SYS\_C00891 INDEX  0   4   152 61440 15**

* If a segment is fragmented, you can rebuild the object into a single segment by using the proper size for the storage parameters.  Export the data for the segment, recreate the object, then import the data into the **INITIAL** extent.

**Defragmentation of Free Extents**

A **Free Extent** is a collection of contiguous free blocks in a tablespace that are unused.

* If a segment is dropped, its extents are **deallocated** and become free, but these extents are **not** recombined with neighboring free extents.
* **SMON** periodically coalesces neighboring free extents only if the default **PCTINCREASE** for a tablespace is **non-zero**.
* The **ALTER TABLESPACE *tablespace\_name* COALESCE** command can be used to force the combining of free extents.
* Your readings will list a number of scripts available in Oracle to test whether or not free space needs to be coalesced.

**Identifying chained Rows**

* **Chained rows** occur when a row is updated and will no longer fit into a single data block.
* If you store rows that are **larger** than the Oracle block size, then you will cause chaining.
* Chaining affects performance because of the need for Oracle to search multiple blocks for a logical row.
* The **ANALYZE** command can be used to determine if chaining is occurring.

**ANALYZE TABLE *Table\_Name***   
**LIST CHAINED ROWS INTO Chained\_Rows;**

* The output is stored to the **CHAINED\_ROWS** table in your schema.  The **CHAINED\_ROWS** table needs to first be created by executing the **UTLCHAIN.SQL** script in the **$ORACLE\_HOME/rdbms/admin** directory.
* If chaining is prevalent (all chained rows are listed), then rebuild the table with a higher **PCTFREE** parameter.

**Increasing the Oracle Block Size**

* Oracle support different block sizes, but the most common block sizes used are **4K** and **8K**.
* Installation routines default to different block size values.  For example, with our version of Oracle in a LINUX environment the system **defaults to 8K** if you do not specify the block size when you create the database.

o   Using the next higher block size value may **improve** performance of query-intensive operations by up to **50 percent**.

o   **Problem**:  You must rebuild the entire database to increase the block size.

o   Improvement comes because the block header does not increase significantly leaving more space in a block for transaction information and for data rows.

**Bulk Deletes: The TRUNCATE Command**

* Deleting all the rows in a table will not save any space because the segment for the table is still allocated all of the extents beyond those the first one that was allocated by the **INITIAL** parameter.
* Deleting all rows can also result in UNDO errors because the bulk delete causes a very large transaction – can lead to overwrites and the **Snapshot Too Old** error message..
* The **TRUNCATE** command resolves both problems, but you need to realize that this is a **DDL** command, not a **DML** command, so it cannot be **rolled back**.
* The **TRUNCATE** command is the fastest way to delete large volumes of data.

**TRUNCATE TABLE Employee DROP STORAGE;**

**TRUNCATE CLUSTER Emp\_Dept REUSE STORAGE;**

* The above command deletes all **Employee** table rows and the **DROP STORAGE** clause de-allocates the non-INITIAL extents.  The second example command is for clusters.

**END OF NOTES**