Oracle8i Tools, Tips and Techniques for Large Data Warehouses

Venkat S. Devraj
IMMEDIENT (formerly Raymond James Consulting)

Introduction

This paper offers broad tips aiding in the physical design and management of large data-warehouses. The tips are a selection of the practices followed at customer-sites worldwide using Oracle. From a physical design perspective, there are two primary characteristics of DSS applications :

- Extensive reads: Such an application would support large number of SELECTs, ORDER BYs and GROUP BYs.
- Large Batch Operations: Most of the data that comes in, is not entered via interactive screens. Instead the source would be operational systems (OLTP and parent DSS applications). The data would be fed in via SQL*Loader processes and custom bulk-load routines (Pro*C, PL/SQL, Perl). Since a lot of data would need to be "cleaned" and summarized, prior to loading, pure database imports would be relatively few (other than for backup operations).

Primary Drivers

This document focuses on performance. Sub-ordinate drivers include recoverability and administrative ease. Generally, recoverability is the primary driver for almost any application. However this is not necessarily true for a data-warehouse application due to the following reasons:

- All the data in the warehouse is derived from existing operational systems. Hence, in a worst-case scenario, they can be completed re-created, using the programs already built. As such, it is highly recommended that any and all programs, built for populating the data-warehouse be designed and developed from a "re-usability" standpoint.
- Data-warehouses are generally not as mission-critical as OLTP systems. They help businesses tremendously. However, they are generally not the "bread and butter" operational applications. As such, relatively more downtime is tolerated.
- Data-warehouses tend to be extremely large in size. As such, MTTR (Mean Time To Recover) is proportionately large. On many occasions, urgently required data can be *re-created* within the same time-frames by re-loading from the operational sources. Such re-loading would save time required to backup the warehouse and resources such as disk-space, tape-drives, backup hardware and software. A prudent approach would be to evaluate the warehouse and classify all segments into different backup groups and frame backup and recovery strategies accordingly.

Segment Groupings

All segments, including tables, indexes, temporary and rollback segments need to be grouped in multiple ways and treated accordingly. Requirements typically determine the groupings. Some sample criteria (there may be additional criteria specific to your site) include:

- Backup requirements: As mentioned above, all segments belonging to a specific backup type need to be physically stored in a similar fashion. That would facilitate backup and recovery operations considerably.

Backup-classification (based on availability)	Backup Strategy
Absolutely Required All the Time (ARAT)	Every night including week-ends
Absolutely Required during Pre-defined Times (ARPT)	Every night including week-ends
Required Most of the Time (RMT)	Weekly
Would-be Good-to-have Most of the Time (WGMT)	Monthly
Can be Easily Re-created (CER)	No backups required

Different backup strategies could be adopted for different segment-types. For example, ARAT and ARPT segments could be backed up just as zealously as regular operational segments (every night). RMT and WGMT segments may be backed up during pre-defined intervals (say, every month). CER segments may not even be backed up (they can be easily re-created in case they are lost).

Usually in most applications, including data-warehousing, a handful of segments dominate the database. Such segments would usually form the ARAT segments.

- Segment sizes: All segments may be grouped into the following types :

	Possible classification criteria		
Size classification	# Rows	Size	Examples
Very Large	> 5 million	> 1 Gb	Fact tables
Large	Between 1 million and 5 million	> 512Mb and <= 1GB	Summary tables, pseudo-operational tables
Medium	Between 10,000 and 1 million	> 100 Mb and <= 512 Mb	Dimension tables, temporary data staging tables
Small	< 10,000	<= 100 Mb	Reference code and domain tables

Segments need to be stored based on their classification. Similar segments can be stored in the same tablespace. However, if they are accessed together, they need to be separated to allow concurrent access by the disk controllers. Also, storing segments with similar size and growth characteristics together, controls fragmentation effectively (since free data-blocks can be re-used).

- Growth: Certain segments might be currently small or medium-sized. However they might have the potential to grow at an extremely fast rate in the short to medium term. For such segments, a small *initial extent* (an Oracle segment grows in *extents*) is usually adequate. However, the *next extent* size needs to be large. Segments can be classified based on their growth. Very large segments that are expected to grow extremely fast (at a pace different from other large segments) can be placed in separate tablespaces. Oracle uses a STORAGE clause (specifying the INITIAL and NEXT extent sizes) to indicate parameters that influence the segment's growth.

Some of the important STORAGE parameter settings for a warehouse (from the perspective of reducing *bubble* fragments, which are practically unusable free blocks, forming "holes" between extents) are :

- INITIAL and NEXT extents should be multiples of the DB_BLOCK_SIZE. There should be a pre-defined number of standard INITIAL and NEXT segments sizes for all segments. The table below lists four such sizes:

Size-classification	Initial Extent	Next Extent
VERY LARGE	10Mb	10Mb
LARGE	5Mb	5Mb
MEDIUM	1Mb	1Mb
SMALL	10K	10K

One thing to note here is, all sizes should be multiples of DB_BLOCK_SIZE. For instance, if the DB_BLOCK_SIZE is 16K, then the INITIAL and NEXT extent-sizes for the SMALL size-classification would be 16K and not 10K. Also, all sizes are relative to the overall database size. The above sizes are just listed as samples. For instance, if your data-warehouse runs into terabytes, it would be easier to classify the VERY LARGE size as 100Mb or even 1Gb, rather than 10Mb. If the objective is to accommodate the entire segment within the first extent, the INITIAL extent should be made a lot bigger. However, in case manual database striping is requiring, then multiple extents may be *pre-allocated* on different disks/controllers (via the CREATE TABLE/INDEX or ALTER TABLE/INDEX commands) with both INITIAL and NEXT being the same.

Note : There is quite a bit of argument among DBAs and Database Architects regarding the adverse effect of a segment having multiple extents versus a single extent. From my perspective, both scenarios have their advantages and disadvantages. For instance, multiple extents allow manual striping (where OS/hardware striping is not done) by allowing extents to be pre-allocated across multiple disks/controllers. However a large number of extents may delay space-management operations (for example, truncating or dropping a large table). Inversely, a single extent allows space-management operations to occur faster, however restricts manual striping flexibility. Depending on your (unique) environment, you need to decide whether you want to allow a segment to occupy multiple extents. With VLDBs, it may be quite a challenge to accommodate a large segment within a single extent. Also with the UNLIMITED option in newer Oracle releases, there is less incentive to do so.

- PCTINCREASE should be set to zero.

- PCTFREE should be set very low (say, 5) for most segments (since direct UPDATEs would be rare). PCTUSED can be set to a high-value, instead, if large-scale DELETEs are prominent. For indexes, PCTFREE can even be zero. This is because in B-Tree indexes, PCTFREE is used only during index-creation and is ignored after that (even if the index-key is updated, since an UPDATE results in a DELETE and re-INSERT within the index).

- Set INITTRANS and FREELISTS to the same value (higher than the default of 1). They should be set to the maximum number of concurrent transactions on the table. For bulk parallel loads, determine the degree of parallelism and set it accordingly.

Top 20 Physical Design Considerations

1. Parallel Query Option (PQO): Set appropriate degree for all large segments. Use PQO to enhance application performance and reduce application downtime, while performing various maintenance operations on application segments (creating/defragmenting tables, rebuilding indexes, etc.). This is particularly relevant to versions prior to Oracle8i (since in Oracle8i, non-blocking index rebuilds and defragmentation is allowed, which allows concurrent DML during these maintenance operations). Also, ensure that tables/processes that are accessed/run simultaneously do not compete with each other, in terms of CPU, disks/controllers and memory and saturate system resources. Severe bottlenecks due to such competition may virtually bring the system to a halt!

2. Cost-Based Optimizer (CBO): Unless CBO is turned on and statistics analyzed after every major load, many newer Oracle features cannot be used. Many of Oracle8 and Oracle8i's sophisticated features such as materialized view usage, query rewrites, star transformation (including bitmap index usage) and so on require the CBO. Regular analyzing only places a Shared-Exclusive row-lock (SX-R) on the table. As such, it can be run during any time of the day/night. However since it is resource-intensive, it is advisable to run it during times of low usage. Ensure that adequate temporary space is available during an analyze. Analyzes sometimes take up temporary space up to four times the segment-size. If possible, always do a COMPUTE STATISTICS, rather than an ESTIMATE. While using indexes on columns with skewed values (example : 500,000 => Females, 10,000 => Males), try and create histograms for the columns.

3. Multiple buffer pools: Place all segments into different buffer pools based on usage patterns. For instance, all large tables, accessed infrequently, may be placed in the REUSE pool, separate from the smaller-tables, which may be placed in the KEEP pool. This will allow the smaller-table blocks to be retained in the buffer cache for a longer time. When the buffer cache is not partitioned into multiple pools, the blocks of large tables, even if accessed infrequently, are placed on the MRU (most-recently used) end of the buffer cache, causing the smaller-table blocks to be moved towards the LRU (least-recently used) end of the buffer cache, thus causing them to be flushed out sooner (assuming that the blocks of the large table are not read via full-table scans; the blocks read off full-table scans are placed at the LRU end, unless the table is cached). When the smaller-table blocks are needed again, they have to be physically read in from the disk. Creating separate buffer pools minimizes such occurrences.

4. Materialized views: During various situations, end-users are forced to run massive queries that summarize large amounts of data and compute certain values based on such summaries. Generally, these kinds of queries are prevalent in databases supporting DSS-type applications. However occasionally, even OLTP databases see these ugly queries being run, thus affecting regular OLTP performance adversely. Oracle8i introduces a new feature called materialized views where the requisite data may already be stored in a pre-summarized fashion, thus allowing applications to directly access and use them, without having to perform any real-time / ad-hoc summarization of the data. The summarized result-sets are made available for single-row retrieval, just like retrieving from a regular table. Materialized views work on the concept of "summarize the data once and use many times", thus saving repetitive data-aggregations by end-users.

   Materialized views are schema objects (like tables, indexes, etc.) that allow data from a base-table to be maintained in a different form within the same database or in the same or different form within a different database. Snapshots (used for replication) fall under this broad category, as well. Materialized views in the form of snapshots are maintained purely for data replication (across distributed databases). However one of the prominent reasons for introducing materialized views in your physical design is to allow applications ready-made access to summarized data. However detailed analysis is required prior to creating a materialized view to determine which are the necessary base tables/columns to be included in the view, what kind of summarizations and aggregations do users typically perform on this data and the best way to present it to them in a ready-to-use format.

   Materialized views are quite similar to indexes in concept, since they would need to be maintained as and when the data in the base tables change (thus potentially causing overhead during writes). In order to prevent materialized views from delaying bulk data-loads to the base tables, you may drop/disable them prior to the load and re-create/re-enable them after the load. Additionally, they tend to consume space. This space consumption may be substantial, if the views are build on large tables, however they save a correspondingly large amount of work. Materialized views can be refreshed completely each time or on an incremental basis. A materialized view log (similar in concept to a snapshot log) is used by Oracle to allow incremental refreshes.

   Materialized views use a table internally (again, just like the snapshots you are so familiar with). You can treat this table just like any other table. In other words, you can created an index on it, you can partition it, etc. (you can also created a materialized view on a partitioned table). Materialized views require the cost-based optimize (CBO). Materialized views can be used explicitly or implicitly by end-users. They can directly query the views (if they are aware of it's existence). In case the application is an existing one and the users are long accustomed to using the base tables directly, they do not have to be re-trained in any way. They can still continue using the base tables in their queries and CBO will automatically rewrite their queries to use the materialized views whenever possible/appropriate.

   Big queries are prone to frequent failure. Think about how often your end-users have ended up getting an error after hours of work ("Snapshot too old" messages, not being able to perform large sorts due to inadequate space in the temporary tablespace, etc.). Also if your site lacks a full-fledged reporting instance, certain class of users are prone to issue large DSS-type queries directly to your OLTP database. Think of the times you had to stop them to prevent regular OLTP transactions from slowing down, thus depriving them the database availability they need. Materialized views enhance availability by allowing the required summary data to be readily available in a pre-formatted fashion. Thus, the end-user does not have to perform any complicated computations of data and critical reports are guaranteed to run in time without any major impact on system resources. The required data is there, whenever they need to access it. These features of materialized views make them almost mandatory in any environment. However a caveat to be borne in the back of your mind is that getting materialized views to be utilized in query rewrites (especially for complex queries) can be highly time-consuming. As such ensure that you are very familiar with the rules pertaining to usage of materialized views in query rewrites (the Oracle documentation lists all these rules).

Furthermore, CUBE and ROLLUP functionality is available in Oracle8i SQL as standard operators, allowing further multi-dimensional (OLAP) analysis on the summarized data in the materialized views.

5. Partitioned segments: Create multiple partitions for large segments. Place popular partitions on separate tablespaces, across multiple disks/controllers. This will reduce contention during bulk-loads and enhance parallel performance (whenever possible, Oracle8 implicitly tries the PQO path, while accessing partitions). Also, placing different partitions on separate tablespaces allows specific (older) partitions to be taken offline during routine maintenance and emergencies, while other (newer) partitions are still online and available for user-access. Such features of partitioning enhance availability. Also, devising a robust partitioning strategy is the most effective way to allow periodic purging of (old) data. For instance with partitioning, it is relatively easy to implement a "rolling window" scheme where data prior to the desired window of time (say older than five years) can be truncated (after archiving it, if necessary). Wherever possible, create local partition key indexes to allow table partitions and their corresponding local indexes to be taken offline/online simultaneously as a single unit.

   Oracle8i allows different partitioning strategies such as range partitioning, hash partitioning and a combination of the two (composite partitioning). Range partitions are highly flexible and maneuverable (allowing for "rolling window" scheme implementations). However in situations where the partitioned data is skewed they may be prone to uneven spread-out of data. Hash partitions come to the rescue in such situations by allowing data to be evenly spread out based on application of hash functions on the partitioning key. However hash partitions are not as flexible as range partitions. In situations where both even spread of data and maneuverability is desired, composite partitioning can be utilized where data may be initially partitioned based on specific ranges and then each range partition can then be sub-partitioned into multiple hash partitions.

6. Transportable tablespaces: The transportable tablespace feature allows you to move/copy a subset of an Oracle database from one Oracle database to another. Transportable tablespaces aid in any situation requiring bulk data to be transferred such as during the following scenarios :

   - Migration of the source data from an OLTP database to a reporting database or transfer of data from operational data sources (ODSs) to the data-warehouse
   - Archival of pertinent data prior to purging it without significant impact to the source database
   - Copying data from the enterprise data-warehouse to temporary staging databases for massaging / converting data prior to loading down-stream data-marts

   Moving data via transportable tablespaces is much faster than conventional unload/load utilities such as export / import and SQL*Loader. The reason for this is, transporting a tablespace only requires the datafiles to be copied across from the source to the target database and integrating meta-data pertaining to the tablespace structure. Index data can also be copied / moved without them having to be rebuilt.

   While transporting a tablespace, it is placed in read-only mode to ensure that a consistent image of the data is captured. Then only specific dictionary information is exported from the source data-dictionary. Next the tablespace data-files are copied across to the target database via any OS utilities/commands (cp, ftp, etc.). Then the meta-data describing the tablespace is imported into the target database. This is very fast because the size of the import is miniscule. Optionally, the transferred tablespace can be then placed in read-write mode. The actual implementation includes using the exp / imp utilities. The exp utility has a new option

TRANSPORT_TABLESPACE while imp has three new options TRANSPORT_TABLESPACE (mandatory), DATAFILES (mandatory) and TTS_OWNERS (optional) to support this feature.

In the current release of Oracle8i, you can transport tablespaces only between Oracle databases that use the same data block size and character set. Also, the source and target platforms should be compatible and if possible, from the same hardware vendor. Also, to prevent violation of functional and physical dependencies and referential integrity, there is another limitation that allows only "self-contained" tablespaces to be transported. Self-contained means that there should be no references within the tablespace pointing to segments outside the tablespace. For example, if the tablespace set being transported comprises of indexes, whose corresponding tables are in different tablespaces, then the source tablespace set is not self-contained. The tablespace set you wish to copy must contain either all or none of the partitions of a partitioned table. The procedure DBMS_TTS.TRANSPORT_SET_CHECK can be used to determine whether or not the tablespaces in question are self-contained. All violations reported by this procedure can be seen within the TRANSPORT_SET_VIOLATIONS view (which will be empty when there are no violations). In order to determine which of the tablespaces in the current database have been transported, the PLUGGED_IN column in DBA_TABLESPACES can be checked.

Additionally, TSPITR (tablespace point in time recovery) can now be used with transportable tablespaces thereby providing more flexibility over TSPITR available in Oracle8.

7. Bitmapped indexes for low-selectivity columns: Create bitmapped indexes on all low-selectivity columns (columns with low cardinality), which are used in WHERE clauses of queries. Bitmapped indexes help read performance drastically. In addition, bitmapped indexes take just a fraction of the space of a regular B-Tree index (in tests I have performed, they allow upto 98% space-savings compared to B-Tree indexes). Also, the new STAR_TRANSFORMATION access-path is possible only if bitmapped indexes exist across the fact/dimension set (here multiple bitmapped indexes are merged using bit-level set operations). A potential worry for database architects is the recommendation that bitmapped indexes are effective on columns with low cardinality. Bitmapped indexes are necessary for STAR_TRANSFORMATION. However if the dimension table happens to have a lot of rows (10,000 or more), then the high cardinality violates the recommendation. In such cases, it is prudent to consider the degree of uniqueness, rather than just the cardinality. For instance, even if the cardinality is 50,000 in a 50-million row table (i.e. 50,000 unique values spread across the 50 million rows), the degree of uniqueness is still high. Creating bitmapped indexes on such columns would still help performance.

8. Reverse-key indexes for lop-sided B-Tree indexes: For columns with high selectivity, B-Tree indexes are often the best solution. Consider reversing the indexes if they are (expected to be) lop-sided (where the number of rows per leaf-block varies drastically). This will reduce large-scale index range scans.

9. Index-organized tables (IOT): All code and reference tables may be created as an IOT. The entire row in an IOT is stored in a B-Tree format (similar to an index). Performance enhances drastically during an access of a cached IOT. With Oracle8i, each row in an IOT has a logical ROWID and the table structure can be altered (just like regular tables). Also, IOTs now support more than one index and can be partitioned.

10. Database block size: Set the DB_BLOCK_SIZE of the production database to 16K (at least 8K if memory is insufficient). Furthermore, set the DB_BLOCK_SIZE of the development and test instances to 8K. This will

allow maximum memory to be provided to the production instance. If a program's performance is acceptable on a test-instance, it is usually better on the production instance, since the latter would have better resources. Setting the DB_BLOCK_SIZE higher would allow more data to fit in one block, thereby helping bulk loads and SELECTs (using large ORDER BY and GROUP BY). More data in each block translates to fewer I/O operations.

11. Full table/index scans: Full table-scans and fast full index scans (FFIS) in Oracle are governed by the parameter DB_FILE_MULTIBLOCK_READ_COUNT. Ensure that the product of this parameter and DB_BLOCK_SIZE equals (at least) 64K. 64K is the OS upper limit on I/O operations (many OSs). This will allow more blocks of a table/index to be read in a single I/O operation.

    Also, enable and use Fast Full Index Scans (FFIS) for large indexes, consisting of all required columns in the SELECT statement. If Oracle finds the necessary columns in the index, it will skip the table access.

12. Redo generation and ARCHIVELOG mode: For very-large loads on DSS-type databases, set NOLOGGING at the segment level. This will reduce redo generation and subsequently, reduce log-switches. In versions prior to Oracle8, UNRECOVERABLE was the only option to reduce redo generation. However UNRECOVERABLE was set at the operation-level. As such, if a third-party tool was used to perform the loads and if the DBA has no control over the tool to perform large loads in UNRECOVERABLE mode, there was no way of reducing redo (unless the database was altered to NOARCHIVELOG mode, which would mean a database bounce and downtime). The Oracle8 NOLOGGING attribute is set at the segment level. As such, no matter what tool is used to perform the loading, redo can still be restricted for certain operations. If NOLOGGING cannot be set prior to huge loads (due to versions prior to Oracle8 being used or if performing non-direct SQL operations), you might have to change to NOARCHIVELOG mode. This will prevent the online redo-logs from being archived, reducing I/O operations. Also, this will prevent the archive-log destination directory from quickly filling up and freezing the database. However, ensure that the new data written is backed up as soon as possible after all NOLOGGING operations or after the database is placed back in ARCHIVELOG mode. Enable database archiving only on the production instance. Note that here I assume that since the availability requirements of DSS applications are not generally as stringent as OLTP applications, you will be able to change ARCHIVELOG modes as a pre and post data-load step (change to NOARCHIVELOG mode prior to the load and revert back to ARCHIVELOG mode after the load) - since data-loads are run on DSS databases when there are no end-users accessing the data.

    For huge bulk inserts, in addition to NOLOGGING/UNRECOVERABLE, use the APPEND hint in the INSERT statement to perform direct-path inserts and avoid the overhead of additional scans for free-space in blocks below the segment high-watermark.

13. Selective resource allocation: System resources such as CPU cycles and memory are always scarce in any large database environment. Data warehouses are not immune to this scarcity. Oracle8i provides a mechanism to control resource usage selectively based on who is accessing the warehouse. To avail of this functionality, the users need to be classified based on the nature of work they are doing and it's importance (for instance, data-entry clerks versus the CFO). Accordingly, resource plans can be created and assigned an appropriate amount of system resources. These resource plans can then be allocated to each user to consume system resources sparingly. Thus, larger and more critical jobs can be provided a bigger resource pool to complete successfully on time.

14. New JOIN-types: Enable SEMI_JOINS, HASH_JOINS and ANTI_JOINS to enhance to speed up queries with EXISTS clause, multiple tables (joins) and NOT IN clause respectively.

15. Constraints: Declare all constraints. Careful analysis is needed to determine whether they can be disabled without violating basic, referential and domain integrity. If possible, keep most of them disabled, so that performance is not hampered. So why create such (disabled) constraints in the first place ? For two reasons : (1) for documentation within the data-dictionary and (2) for usage by ad-hoc query tools that use such info for determining join-criteria.

    Determine whether all enabled constraints can be deferred. If so, defer them for the duration of the entire load. If the load is very large, then check whether the load can be broken up into pieces, parallelized and inserted with the constraints deferred.

16. Rollback Segments: Create medium and large sized rollback segments. However keep the medium-sized ones disabled and enable them only when a large number of jobs need to be run simultaneously. Do not reference medium-sized ones in ROLLBACK_SEGMENTS *init.ora* parameter (so that they remain disabled even after an instance bounce).

17. Clusters: Oracle clusters allow pre-joining of related tables based on an indexed cluster-key or a hash algorithm. The main disadvantage with clusters is their lack of flexibility. This flexibility is more pronounced for OLTP applications. However a data-warehouse could benefit substantially from a cluster. Much manual denormalization can be avoided by using clusters. Consider index and hash clusters for medium-sized tables. However, small and large tables are best left alone. The reason is : small tables are really insignificant, as far as storage is concerned. By placing them in a cluster with a large table, a significant amount of disk-space could potentially be wasted (due to constant repetition of a small number of values). In the case of large tables, a lot of flexibility is required for organization and maintenance before and after large data loads. This flexibility would be lost due to clusters. The best candidates for clusters are medium-sized tables, which are related to one another and are almost always accessed together.

18. Denormalization of fact tables: Do not denormalize fact tables extensively (i.e. carry a number of non-key dimension columns within the fact). Instead, make use of the Oracle8 STAR_TRANSFORMATION path. Besides hampering performance at times, extensive denormalization of fact tables also result in high wastage of space. Joins are not always bad. In fact, under certain conditions (when a smaller dimension table is used to reduce the number of rows retrieved off a large fact table as in STAR_TRANSFORMATION), they can indeed be faster.

19. Large tables and indexing: Do not index large tables extensively. The common perception is that in a data-warehouse, indexing is dominant. Yes, to an extent that is correct. Compared to an OLTP application, indexing is far more prevalent in a data-warehouse. However with indexing, it is easy to get carried away. Analyze *potential* usage patterns carefully and prepare a list of candidate columns in each large table requiring indexes. In the first run, index them all. Once the warehouse is deployed, monitor it for *actual* usage patterns. The index list needs to be re-visited at that point and all redundant indexes need to be removed.

Extensive indexing can be a nightmare for both DBAs and end-users due to the additional maintenance requirements. If an index is not well-maintained, it can easily degrade performance. Data-loads on tables with large number of indexes can be extensively slow, compared to a table with a few or no indexes. Sometimes, in order to enable large data-loads to complete on time, it is essential to drop the indexes (or mark them UNUSABLE) prior to the load and re-created them later. If the table is very large, index re-creation can take a lot of time, sometimes more time than the load itself! As of Oracle8 8.0.x, multiple indexes for a table cannot be created in parallel since each index creation statement requires a table lock.

20. Truncate time on date columns : On date columns, if the time portion is not absolutely necessary, store only the DD-MON-YYYY portion. When indexed, this will allow equality searches (WHERE date_col = <value>) to be used more often, helping performance. In case the time part needs to be retained, consider denormalizing the table by introducing a second column to store only the DD-MON-YYYY (in addition to the first date/time column). This second column may be indexed to enhance equality searches (the probability of using an equality search reduces drastically if the time is also stored).

A handy tool to facilitate physical design

The following list provides a (table-oriented) physical design checklist to ensure that all related aspects are considered and documented during the logical-to-physical design conversion stage (especially for the segment-groupings mentioned above). A spreadsheet may be used to facilitate usage of this list.

Form : TP-1 Table details	
Table name	
# of columns	
# of constraints	
# of triggers	
# of LOBs	
Source-of-record	
Partitioned	
# of partitions	
Partition range	
# of records in initial data-load	
Frequency of data-loads	
# of records in each subsequent load	
Parent tables	
Child tables	
Other tables accessed with this table	

Form : TP-1 Table details	
Table name	
Full-table scans (%)	
Indexed-lookups (%)	
For indexed-lookups, would all columns be used (SELECT *, WHERE..., GROUP BY ..., ORDER BY...)	
Purge Frequency	
Archival to off-line storage necessary before purge (Y/N)	
Archival retention-period	
Tolerable down-time	
Time required to re-create table from source-of-record	
Heavy access periods	
# of concurrent transactions during heavy access periods	
Non-access periods	
Average # of concurrent transactions	
Average transaction size	
Transaction % that can select a rollback-segment	
Table format (Index-Only / Regular)	

Table name	
Column name	
Data type	
Length	
Scale	
Null-value allowed	
Source-of-record formulae	
Average size	
Access frequency	

Table name	
Column name	
(U)nique / (N)on-unique	
Density	
# of constraints	

Table name	
Column name	
Constraint name	
Constraint type	
Constraint columns (in proper sequence)	
Referenced table name	
Referenced columns (in proper sequence)	
Constraint condition	
Permanent Status	
“During load” status	
Deferrable	

Table name	
Index name	
Index type (Bitmap / B*Tree / Reversed)	
(U)nique / (N)on-unique	
Columns in index (in proper sequence)	
Data already-sorted (Y/N)	
Partitioned (Y/N)	
# of partitions	
Partition range	

Table name	
Trigger name	
Trigger event	

Table name	
Trigger name	
(E)vent / (R)ow level	
WHEN condition	
Trigger body	

Table name	
LOB column name	
LOB type	
LOB content type	
Average LOB size	

Summary

The paper recommended that all segments be classified based on specific criteria. Backup/recovery requirements, segment sizes and growth frequency were mentioned as sample criteria for such classification. This classification would form the basis for physical storage (both within a tablespace and within the SGA) and other essential tasks such as backup and recovery. A few tips, essential for enhancing performance were outlined. And finally a tool acting as a checklist for consideration and documentation during the logical-to-physical design conversion stage was introduced.

References

- Oracle8i Server Documentation
- Oracle 24x7 Tips and Techniques; Venkat S. Devraj; Oracle Press (Osborne/McGraw-Hill); 1999