Optimal Physical Database Design for Oracle8i

Dave Ensor
BMC Software, Inc

The opinions expressed in this paper are those of the author, and are not necessarily shared by BMC Software, Inc.

Introduction

Oracle8i adds a number of significant new features in the areas of indexing and space management along with major upgrades to the support for Index Organized Tables (IOT's). The paper presents performance figures on index compression and IOT's, and analyzes the performance and operational impact of online table reorganization, which is now supported for IOT's. A strong case is made for migrating certain types of table to IOT's though, as discussed, the change may not be transparent at the application level.

The paper also looks at temporary tables, and explains how they can both improve performance and reduce one specific type of application failure. The circumstances under which temproary tables should be used are detailed along with application changes that should be considered. Performance improvements achieved from both locally managed and transportable tablespaces are also presented, and the potential implications are explained.

Physical Database Design 101

Physical Database Design is a large and complex subject, but this section sets out to cover the issues that most commonly require consideration when planning the physical structure of an Oracle database.

Because of the increasing use of disk striping and storage array controllers, this paper assumes that I/O load balancing can be achieved without the direct involvement of the Oracle DBA and does not discuss the placement of the container files used to store an Oracle database. In this context it may be worth noting that journaled file systems such as the Veritas File System have been shown to yield better Oracle performance than the standard UNIX file system.

What is Physical Database Design?

Most of the design decisions that have to be made when creating an Oracle schema are logical rather than physical, and concern the logical definition of tables and their columns, and views and their columns. An Oracle DDL statement such as

```
create table STOCK ( PART#    number       not null primary key
                   , QUANTITY number       not null
                   , LOCATION varchar2(20) not null
                   ) tablespace DATA01 pctfree 15 pctused 0;
```

contains both logical and physical elements. The table and column definitions are logical and some knowledge of them will be required to write queries against the table whereas the space management clauses are purely physical and no knowledge of them is required to perform data operations against the table. By this argument the DDL statement

```
create index STOCK_LOCATION on STOCK (LOCATION)
                               tablespace DATA01 pctfree 50;
```

is pure physical design. It cannot affect the *result* of a query or DML operation against the table even though it may have a radical effect on the performance of the operation.

Block Structure

It is important to realize that rows in an Oracle table are almost always true variable length, and that the row length typically changes with each update. The most common exception is a table all of whose columns are of datatype `CHAR` or `DATE` as this data is stored fixed length. However a common use of the SQL `UPDATE` command is to replace a `NULL` value, and in this case the row is guaranteed to expand in length. To allow for row expansion, tables that are subject to updates should be created with a value for the storage parameter `pctfree` that leaves enough space in each block for foreseeable row expansion. The default value of 10 (which means that insert will leave every block 90% full or less) is rarely ideal and tables that are only subject to insert and delete should always specify `pctfree 0` to optimize space allocation.

If random rather than bulk deletions are performed against the table then it is also worth considering the value for `pctused`. This specifies the point at which the block will again become available for data insertion. The default value is 40, meaning that blocks are available for row insertion when they are less than 40% full. A *freelist* mechanism identifies the blocks available for insertion, and freelist maintenance carries a CPU and disk I/O penalty. In general the lower the sum of `pctfree` and `pctused`, the less freelist maintenance will take place. As this sum approaches 99 (the maximum permitted value) a series of negative effects will be observed, and DBA's are strongly recommended to specify `pctused 0` wherever possible. On the other hand if a row expands and there is no longer sufficient space for it in its original block, then the row is *migrated* to another block and this causes a performance penalty on any indexed retrieval of the row. For a table with any significant update rate and a high indexed retrieval rate `pctfree` should be set relatively high as the increased efficiency of indexed retrieval will outweigh the penalty of a slightly larger table.

The block structure of B*tree indexes is broadly similar but far from identical. In this case `pctfree` is used only during index creation, and leaves distributed free space in index leaf blocks. If new keys have random values rather than always being higher than the current highest key then index space management during DML operations can be all but eliminated by rebuilding the index at regular intervals with distributed free space. To allow for a doubling of index size, the index would be built with `pctfree 50`. If, on the other hand, every new index key is higher than the previous highest key then the index should be built specifying `pctfree 0`.

Block Size

There have been a number of papers at recent Oracle conferences describing the advantages of using a database block size greater than the traditional standard of 2048 bytes.

DBA's are strongly recommended to create Oracle databases with a block size of 8192 bytes except where there are compelling arguments for use of a different size. Larger block sizes reduce the number of spanned rows (rows that cannot fit in a single block) and save disk space in all but the smallest tables because less of the disk is used for the gaps at the end of each block. This disk saving in turn speeds up full table scans. Increasing the block size will also reduce the height of the tree for many indexes, and speed up index lookup.

Unstructured data

Until Oracle8 the only mechanisms for storing large units of unstructured or encapsulated data were the `LONG` and `LONG RAW` datatypes. These have a number of functional disadvantages, and they also store the unstructured data inline in the row. This significantly slows full table scans, and can also cause long chains of row pieces that have to be navigated even during processing that requires access only to the structured data. In Oracle8 the LOB datatypes allow unstructured data to be stored a separate segment with its own storage parameters. This has significant performance and storage management benefits, but unfortunately converting a schema from `LONG` to `LOB` datatypes requires non-trivial code changes, and many tools do not support the LOB datatype.

Freelists

When Oracle needs a new block into which to insert table data, it checks the table's freelist and takes the block at the head of the list. If there are no blocks on the freelist it advances the *high water mark* (HWM), which records the last block which has ever contained data. If there are no blocks left beyond the high water mark then more space must be allocated to the table. This mechanism works well for small numbers of users inserting into the same table, but eventually the number of users sharing the same insert block causes serialization problems (they start having to queue to use the block). This can be detected by checking buffer busy waits for data blocks, and the solution is to recreate the object using the storage option `freelists` to add additional freelists.

In an Oracle Parallel Server (OPS) environment, tables that will be subject to inserts from more than one instance should be created with the storage option `freelist groups` to ensure that database instances do not have to share insert blocks.

Extents

In Oracle every data dictionary object that requires storage owns a storage *segment*; this in turn consists of one or more *extents* each of which is a group of logically contiguous[1] database blocks. All data blocks are the same size, but extents may be any number of blocks up to the capacity of the datafile (or raw device) in which the extent resides. Each segment must exist solely within a single tablespace, but it may extend across multiple datafiles or raw devices within that tablespace.

Extent sizing can be specified at both the tablespace and segment level using the storage parameters `initial`, `next` and `pctincrease`. Although extents can be any size, it is strongly recommended that every extent in a tablespace should be the same size. This is best achieved by setting the default `initial` and `next` for the tablespace to the same value, setting `pctincrease` to zero, and never specifying these parameters at segment or object level. The result is that classic tablespace fragmentation becomes impossible, as every free extent in the tablespace should be either the same size as the requested extent or a multiple of it.

Many DBA's are concerned that this practice will cause some objects (segments) to have an excessive number of extents. This raises the interesting question as to how many extents might be regarded as excessive. Provided that extents are a multiple of the multiblock read count there is no evidence of any performance effect from having multiple extents other than the load of allocating and deallocating the extents. Using Oracle 8.1.5 under NT Workstation 4.0 on a 366Mhz Pentium II with 256Mb of RAM, extent allocation took about 12 msec and extent deallocation about 5 msec. Over the life of the average table this load is trivial even for 1,000 extents. Despite this, the approach of using uniform partition sizes is normally associated with an arrangement where tablespaces are grouped by segment size rather than by object association. This issue is discussed further under Transportable Tablespaces below.

Space management within the `SYSTEM` tablespace should be left entirely to Oracle, and no user objects should ever be created in this tablespace.

AutoExtension

When a segment requires a new extent, and there is no free extent in that tablespace that is equal to or greater than the number of blocks requested, then the user receives an error. This can be partially overcome by allowing at least one of the files comprising the tablespace to *autoextend*. If this property is set then Oracle tries to enlarge the file

[1] It is up to lower levels of software and device controllers to determine whether the blocks are physically contiguous.

by a specified amount until the file either exhausts the space available in that file system, or reaches a preset maximum length. This mechanism is highly valued by some, and totally deprecated by others.

Where a mount point or file system contains data files for many tablespaces, and the DBA is unable to predict which of these will run out of space first, then there may be some benefit in allowing the individual tablespaces to compete for the remaining space. However it is recommended that adequate free space should be preallocated to each tablespace used by any mission critical application, and that active monitoring be performed to predict space exhaustion before it occurs.

Partitioning

Oracle is entirely capable of managing tables of several hundred gigabytes, comprising hundreds of millions of rows. Performing maintenance operations such as bulk deletion, backup or index creation on such tables is challenging, especially in environments where maintenance windows are restricted. The requirement to have each segment within a single tablespace means that there must be a tablespace at least as large as the largest segment, and this poses real space management problems on most platforms.

The solution is to *partition* the logical object into many physical segments, splitting it up on the basis of a *partition key* comprised of one (or more) table columns. In Oracle 8.1 this can be done on the basis of key ranges or by a hash value based on the key. For exceptionally large tables it may make sense to first divide the table into a series of key ranges, often based on date, and then to subdivide these key ranges using hash partitioning.

Both tables and indexes can be partitioned, and an important feature of partitioning is that although every partition of an object must have the same *logical* structure, they may have different *physical* segment properties. Thus the bulk of the partitions of a history table can be directed to read only tablespaces on the grounds that past history may not be updated, whereas more current records can be placed in tablespaces that are available for writing. This approach can dramatically reduce regular backup times and backup volume.

Partitions, and especially date-based partitions, also offer highly efficient bulk deletion to partition-aware applications through the SQL DDL statement

```
alter table … drop partition …;
```

This is especially attractive for partitioned tables with *locally partitioned indexes*. These are indexes where each index partition refers to one and only one table partition. This arrangement allows table partitions to be dropped and new table partitions to be added without any need to maintain a table-level index. The downside of this arrangement in OLTP applications is that unless the index key contains the partition key, then on index lookup every index partition must be visited. For a unique key lookup on a table with 1,000 partitions this would incur an overhead of several hundred to one when compared with a global index on the unique key (whether or not this global index was partitioned).

Global indexes can take considerable time (and enormous amounts of temporary segment space) to build, and become invalid or unavailable if any partition is removed or is inaccessible. However they offer the only efficient means of retrieving low numbers of rows from a very large table when the partition key is not among the criteria.

Index Compression

The bottom level of any B*tree index is the *sequence set*, an ordered list containing each key value with a pointer to the row that contains the key. In all previous versions of Oracle this “pointer” has been the rowid, though in Oracle8i the special case of Index Organized Tables requires a rather different convention, discussed later in this paper.

Although this ordered list was highly compressed in Oracle Version 5, more recent versions have stored every instance of every key in full and this can consume significant disk space. Oracle8i allows indexes with concatenated keys to be built with compression on a specified number of leading key columns e.g.

```
create index SAMPLE_WORDS
          on SAMPLE (WORD1, WORD2, WORD3, WORD4, WORD5)
          nologging compress 3;
```

In the testing performed for this paper, compression was always allowed to default to the maximum number of columns permitted (which in turn depends on whether or not the key is unique).

The author's experience of compressed indexes was almost universally positive. They saved significant amounts of space and were slightly faster to create than their uncompressed equivalent, presumably because there were less blocks to write. No significant performance differences were measured retrieving from compressed and uncompressed indexes, although time did not permit the testing of long index range scans. These were expected to favor compressed indexes because less index blocks would require to be visited.

A deliberately severe update test resulted in a 46% increase in CPU activity over the same test when applied to a table with an uncompressed index, but the increase in elapsed time was almost insignificant. This test involved updating the 3rd column of a 5 column compressed index, forcing the index entry to be deleted and moved to another part of the sequence set. No I/O penalty could be detected during this test.

Index Organized Tables (IOT's)

Tables of Organization Index were introduced with Oracle8, but had a number of restrictions that made them generally unattractive. In Release 8.1 most of the restrictions have been removed, and this special type of table looks to have become a realistic design option. The DDL to create them is straightforward, e.g.

```
create table SAMPLE6
           ( ID#
           , constraint SAMPLE6_PK primary key (ID#)
           , CODE
           , …
           , SUBCODE
           ) organization index pctthreshold 20;
```

Put at its simplest, an IOT is a primary key index acting as a table. If you look in Oracle's online data dictionary, the table exists in `SYS.TAB$` but it has no matching entry in `SYS.SEG$`. An index segment, with the same name as the primary key constraint, is used to store the "table". For this to be effective the sequence set of the index has to be capable of storing non-key columns along with the key columns. As a result no entry in the sequence set may exceed half the block length. This restriction is required because a B*tree must be able to hold a minimum of two keys per sequence set block. When defining an IOT the user may specify the maximum sequence set entry size as a percentage of the available space in each block – the default maximum is 50%. Any data over this size (`pctthreshold`) is stored in a separate overflow segment.

The claimed advantages of IOT's are space savings (the primary key is only held once) and faster access because having located the sequence set entry Oracle has also located all of the column data *unless that data is in an overflow segment*. An IOT also breaks one of the "golden rules" of relational data by storing the data in a guaranteed order though it is risky for an application to rely on this property. If the application requires data in a specified order than that data should be retrieved using an `ORDER BY` clause.

With Oracle8i IOT's may have secondary indexes, but a potential problem arises here. The table rows are sequence set entries, and their position can and will change as other keys are added and deleted around them. Oracle has implemented a simple solution to this problem – a secondary index on an IOT stores the primary key of the target row rather than its rowid. Optionally a pseudo rowid may also be stored to allow more direct navigation

to the target row, though over time this can become inaccurate and the navigation will revert to using the primary key.

Insert Times

Because the index structure has to be built during row insertion, and the rows correctly positioned in the sequence set, it has always been clear that inserting into an unindexed conventional table will be faster than loading into an IOT. On the other hand, tables of any size normally have at least a primary key index and therefore the total time to insert rows into the table must include the creation or maintenance of this index.

Tests were performed to compare the insertion of 100,000 rows into an IOT with the insertion of the same data into a conventional table followed by applying a primary key constraint to build an index. In the first test the primary key was long (WORD1, WORD2, WORD3, WORD4, WORD5) and although the conventional table loaded much faster, this advantage was lost in the time taken to build the index. The space saving was, as expected, massive.

Performance with long primary key (all times in seconds).

Table Type	Insert Time	Index Time	Total Time	Table Blocks	Index Blocks	Total Blocks
Conventional	11	85	96	986	768	1,754
IOT	86	-	86	-	1,040	1,040

When a very short key was used on exactly the same data, a rather different picture emerged. The figures for the IOT barely changed at all, the conventional table was much faster to index, and its space overhead was reduced. The only way found to markedly reduce the insert time for the IOT was to present the data in key order, which removed the need to insert keys into the middle of sequence set blocks. Even in this case the insert time for the IOT was about 40% longer than the sum of the insert and index times for the conventional table.

Performance with short primary key (all times in seconds).

Table Type	Insert Time	Index Time	Total Time	Table Blocks	Index Blocks	Total Blocks
Conventional	11	32	43	986	294	1,280
IOT	85	-	85	-	1,040	1,040
IOT (in key order)	60	-	60	-	1,040	1,040

Retrieval Times

The results from the retrieval tests were less marked than had been anticipated. Retrieval by primary key from an IOT with no overflow segment was faster than retrieval from the equivalent conventional table with a primary key index, and retrieval through a secondary index on an IOT was slower than retrieval through a secondary index on a conventional table. The differences in I/O traffic (more correctly block visits) were consistent with this model.

The overhead of using the secondary index on an IOT was not excessive. However it was felt that because retrieval via a non-unique index typically leads to more rows being retrieved per query than using a primary key

index, retrieval from an IOT via a secondary indexes might amplify the negative performance impact in a production environment.

Application Impact

For almost all purposes an IOT has precise functional equivalence to a traditional Oracle table, now referred to as a Heap Organized Table. However no table used by existing applications should be converted to an IOT without first checking whether and how that application uses rowid's. Represented in character form, the rowid for an IOT not only requires more space (42 bytes as against 18 for a conventional table) but also changes if the primary key is updated. This latter behavior will only affect an application that updates twice using the same rowid, once to update the primary key and then again to update any column in the row. Such behavior is deprecated.

Online Table Reorganization

With version 8.1 it is possible to rebuild or reorganize an Index Organized Table in parallel with normal use of the table, including DML. The minimal version of the syntax is delightfully simple e.g.

```
alter table SAMPLE move online;
```

The keyword `online` is optional. If it is not present then DML against the table is blocked for the duration of the operation but during timing tests it was found that in the absence of any update traffic `move online` was consistently around 30% faster than plain `move`. The reasons for this anomaly are unknown.

The great concern was that the time to perform the move, or rebuild, operation would be greatly increased if updates occurred in parallel with the `alter table … move online` but tests showed that the performance was surprisingly good. A compute bound update loop was coded in PL/SQL to perform 10,000 updates and commit after each update. Run on an otherwise empty machine this took 74 seconds to complete. The rebuild, run without any other load, took 21 seconds for a total of 91 seconds to perform the two tasks serially.

When the two tasks were run in parallel by starting the update and then immediately starting the rebuild, the total elapsed time for the update was 127 seconds with the rebuild running in parallel for 119 of those seconds.

A series of further tests were performed at lower update volumes. These demonstrated that updating in parallel with a rebuild or move approximately doubled the time taken per update and slowed the rebuild by almost exactly the time taken by the update. From the tests performed `alter table … move online` appears to impose a reasonable overhead and to scale well.

As stated above, these update tests were performed in transactions that contained only a single update. It should comes as no surprise that execution of `alter table … move online` requires two "quiet points". Execution of the statement will neither start nor complete while there are uncommitted transactions against the table; while it is underway any number of transactions may be initiated against the table, but they must all complete before the rebuild will end. This is unlikely to be of concern unless the application contains very long running transactions.

For applications that want to get closer to 24*365 but expect to have a need to move or reorganize tables, the availability of `alter table … move online` presents a further motivation to consider the use of Index Organized Tables.

Temporary Tables

Many applications use the database to handle arrays of transient working data. This is especially common in applications written using Rapid Application Development (RAD) tools that may not feature the robust memory management features required in order to handle large data arrays.

The use of fully persistent database objects (tables) to hold transient session-specific data can cause a number of performance and functional problems in an Oracle environment. All changes to such tables are recorded in both rollback segments and the redo log for recovery purposes, and in multi-user environments which share a permanent table for transient use it may be necessary to index every row inserted using the *session id* (SID). Shared tables also require the application to perform expensive deletes at end of transaction or session, and to implement recovery code to clean up after failed sessions. An alternative approach is for each session to create its own "temporary table" but there are a number of performance implications in this approach, and it does not scale well.

Oracle 8.1 introduces the statement form:

```
create global temporary table …
    (col1  …
    , …
    ) on commit preserve/delete rows;
```

Any row inserted into such a table is visible only within the transaction that inserted it unless the qualifier `on commit preserve rows` is present. In this case the row continues to be visible to the creating session after `commit` and until deleted (or end of session). Global temporary tables are therefore a valid design option for any table whose data is never required to be persistent beyond the end of the transaction or session that inserted it.

These temporary tables are allocated in temporary tablespaces, and observation indicates that a single temporary segment is used in each temporary tablespace for all temporary tables for all sessions that use that temporary tablespace. The temporary segment was not observed to shrink as temporary rows were automatically deleted at end of transaction or end of session, but it was apparent that the space released was being made available to new transactions. The only control over the tablespace used for temporary tables is through the SQL statement

```
alter user … temporary tablespace …;
```

Bulk insertion into an unindexed temporary table was about twice as fast as into the equivalent persistent table, and generated about 95% less redo log entries. More surprising, full table scans appeared to be about 25% faster. Temporary tables may be indexed, but this was not tested during the writing of this paper. It should be noted that DDL may not be performed on a temporary table if it contains rows for *any* session.

Application Impact

Many applications that use persistent tables as session workspace are subject to error conditions that bypass the removal of entries made in the table. This problem can exist in any application that preserves the temporary data across more than one transaction, as the data will persist unless specifically deleted. The problem may be especially severe where the session creates its own table to use as workspace, because automatic transaction rollback on error will never undo a DDL operation. The result in both cases is that application failure can leave "temporary" rows in the table, and these may cause future sessions using the table to get incorrect results. The author has seen a number of applications that contained defensive code to recover from rows accidentally left by an earlier session. A major advantage of global temporary tables is that this scenario cannot occur.

In most cases an application can have a work table switched from a conventional table to a global temporary table without code changes being required. This should result in some performance improvement, and will remove the risk that a failed session will leave persistent rows in the database. However most applications will require modification to fully leverage global temporary tables. The changes are typically quite straightforward in that they are mainly concerned with the removal of logic that is no longer required. Examples of functionality that can be removed include table create and drop for applications that built a table with a unique name per session, row deletion at end of transaction or session, daemons to tidy up after sessions that have failed to delete their rows, and indexing by session identifier. None of these are required when using global temporary tables.

Locally Managed Tablespaces

Conventional Oracle tablespaces have their space allocation recorded in the data dictionary in the tables SYS.FET$ and SYS.UET$. Oracle 8.1 introduces an alternative, which is for the tablespace to contain an allocation bitmap for a series of equal size extents. The statement

```
create tablespace SAMPLE
datafile 'D:\oracle\oradata\…'  size 1000M
extent management local uniform size  100K;
```

creates a 1 gigabyte tablespace in which *every* extent will be 100k bytes. Objects can be created specifying extent sizing, but these parts of the DDL are quietly ignored. For DBA's who like the idea of tablespaces with consistent extent sizes to prevent fragmentation, locally managed tablespaces provide the mechanism to fully enforce this approach. Unfortunately they also invalidate any space usage reporting scripts that rely on SYS.FET$ and SYS.UET$. Such scripts must be recoded to use SYS.X$KTFBFE and SYS.X$KTFBUE for locally managed tablespaces. The columns in these virtual tables map easily to the column names in the equivalent data dictionary tables, and the additional virtual table SYS.X$KTFBHC summarizes free space with one row per datafile. There are no GV$ or V$ views to externalize these virtual tables.

In a series of tests Locally Managed Tablespaces appeared to work well and to use significantly less resource for space allocation and deallocation. Traditional tablespaces were found to take more than three times as long to allocate an extent, and more than twice as long to deallocate. Impressive though these figures are, they are only significant to applications that perform altogether too many space management operations (possibly the creation and dropping of temporary tables). Nevertheless the improvements should be of major benefit to instances suffering from type ST lock conflicts. The author would argue that these can invariably be resolved by application design change, but this option is not available to many (most?) system administrators.

Transportable Tablespaces

It is a common requirement to migrate data from one Oracle database to another. Two principal mechanisms have been used in the past. Either the data was transferred using Oracle's distributed database support or it was unloaded from one database, a file physically transported to the destination, and the data reloaded. Both approaches have strengths and weaknesses, and both consume considerable resources and take significant processing time for large tables. In Oracle 8.1 one or more tablespaces may be copied and "plugged into" another database subject to a set of restrictions; in addition a single physical copy of a read only tablespace may be simultaneously part of many physical databases. One of the major restrictions in both cases is that the source and destination instances must be running on the same hardware and OS platforms, and must be the same Oracle release.

Although not strictly part of space management, transportable tablespaces have a number of implications for space planning in an Oracle 8.1 environment because of the mechanism requires *self-contained* sets of tablespaces. Put simply, these are sets of tablespaces where all of the partitions and indexes are present for every table or cluster in the tablespaces. Optionally the definition may be extended to include all targets of referential integrity constraints. This is turn draws into question the use of specific tablespaces for a size of object, and makes more attractive the traditional approach of allocating tables and indexes to tablespaces on a functional basis.

The portable tablespace mechanism requires that the source tablespaces be read-only for the duration of the data copy phase of the operation, and therefore the source instance cannot be said to maintain 100% availability. Nonetheless with careful allocation of objects to tablespaces, transportable tablespaces can offer a highly efficient data transfer mechanism. They also provides an additional incentive for use of the autoextend mechanism to avoid the need to preallocate space to a tablespace, because unused datablocks within a tablespace simply add to the time taken to copy the datafiles and therefore to the time for which the tablespace must be read-only.

In a simple experiment under NT 4.0 on a 366 MHz Pentium II moving a tablespace containing 18 objects, the export processing on the source database took under 20 seconds, and the processing required to plug the tablespace into the destination database took less than 5 seconds. Copy time will, of course, vary with tablespace size and hardware speed.

The major negatives found with the feature were that it was liable to “finger trouble” when transporting multiple tablespaces, and that it was necessary to enter the SYS password *from the keyboard* on each instance in order to operate the mechanism.

Conclusions

Previous good practice in space management remains valid with Oracle 8.1, especially the notion of equal extent sizes throughout a tablespace. Locally managed tablespaces allow this approach to be enforced and appear to have performance advantages though few sites should be performing enough space allocation to see a significant change in overall performance as a result.

Database block size should be 8192 bytes in most cases, and almost never 2048 bytes.

Care in selecting the values of `pctfree` and `pctused` for tables, and `pctfree` for indexes, will be rewarded by improved performance and reduced need to reorganize.

Index compression is recommended for concatenated indexes as a way of both reducing index create time and saving space. No penalties were discovered.

Index Organized Tables are worth considering for any large table whose primary key is a substantial part of the average row, and for tables that may benefit from online reorganization. With either of these conditions satisfied, the less secondary indexes are used to access the data, the stronger the motivation to use an IOT.

If applications *must* use the database as working storage, then global temporary tables have significant performance advantages and will allow the application to be simplified and made more robust in many cases.

Portable tablespaces look to be a promising feature, but one that may require reappraisal of the way in which objects are allocated to tablespaces.