

# How to Win Friends and Influence DBAs

Leslie M. Tierstein,  
WR Systems, Ltd.

## Overview

Application tuning should be a joint effort of the application's developers – who are intimately acquainted with the application database's structure and usage – and database administrators (DBAs), who know just what aspects of the database's physical characteristics can be adjusted to enhance performance and database integrity. This paper examines those aspects of database configuration that can be specified via Designer/2000. At each phase in the development life cycle, developers and DBAs can work together to build an efficient, smoothly running database (and application).

## Analysis Phase Data Collection

The analysis phase is not too early to start collecting information which will affect the physical storage of the application data. This information, which Designer/2000 calls "Volumetrics", can later be used to perform database sizing calculations. Volumetric data can be collected about both entities and their attributes.

### Entity Volumetrics

Designer/2000 can store each entity's initial volume (number of rows); maximum volume; average volume; and annual growth rate, as shown on the Definition tab of the Edit Entity dialog of the Entity Relationship Diagrammer:



The screenshot shows a dialog box titled "Volume" with four input fields: "Initial", "Maximum", "Average", and "Growth Rate". Each field is represented by a rectangular box with a small arrow on the right side, indicating it is a dropdown or text input field.

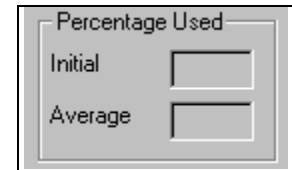
Personally, I've never seen an entity definition that had these filled in. But it's not because the information is unavailable at this phase of the project:

- ?? **Initial volume** - If the project includes a legacy system conversion, the conversion team needs to know the current volume of legacy data in order to develop a conversion and deployment strategy.
- ?? **Maximum volume** - Analysts need to be able to size the database and, potentially, new hardware for the system. To do this, they need to know how much data will end up in the database.
- ?? **Annual growth rate** - Analysts typically ask users how many transactions will be performed per year, to get an idea of how their business works.
- ?? **Maximum volume** - can be calculated once you know the initial volume, projected growth rate, and anticipated system life span..

So, from the project management standpoint, the development standards should specify that volumetrics data must be collected and entered in the repository.

### Attribute Volumetrics

At the attribute level, Designer/2000 allows you to record the percentage of time a particular attribute will be filled in on average and when a row is initially created, as shown in the Att(ribute) Detail tab of the Entity Relationship Diagrammer:



The screenshot shows a dialog box titled "Percentage Used" with two input fields: "Initial" and "Average". Each field is represented by a rectangular box with a small arrow on the right side, indicating it is a dropdown or text input field.

Both the initial and average percent used are automatically set to 100% for attributes that do not allow NULLs, that is, required fields, and to 50% for attributes that may be left blank. If possible, try to get developers to ask the users questions like, "Do *most* of the administrative law cases have a reason for being closed?" If the user comes up with a percentage, use it; if not, use the default. There will be time to collect better approximations during detailed design, and after you can examine a sample of legacy data.

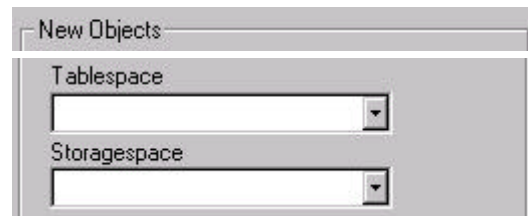
## Database Design Wizard

The Database Design Wizard (DDW) transforms the logical model specified during analysis into a physical model. It transforms entities, their unique identifiers, and relationships into table definitions with primary and foreign key constraints. Doing a modicum of research and preparation before running the Wizard may save an appreciable amount of retrofitting later.

### Preparation

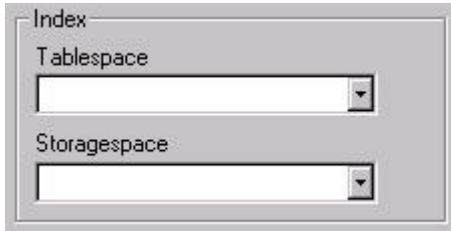
The Options tab of the Database Design Wizard shows the type of preparation required. In particular, the tablespaces and storagespaces to be assigned to tables and indexes have to be thought out, and corresponding definitions created.

The tablespace and storagespace names specified in the New Objects group on the Options tab will be included into every table definition created when the DDW is run.



The screenshot shows a dialog box titled "New Objects" with two input fields: "Tablespace" and "Storagespace". Each field is represented by a rectangular box with a small arrow on the right side, indicating it is a dropdown or text input field.

Similarly, the tablespace and storagespace specified in the Index group will be used to define indexes and primary and foreign key constraints, which use indexes.



### Tablespaces

Here, I run the risk of drastically oversimplifying, since I'm going to be talking about (gasp) hardware.

At the physical level, an Oracle database is made up of a set of files. One or more files comprise a particular tablespace. When you create a table, it is assigned (either explicitly, via the CREATE TABLE command, or implicitly, by your user default) to a particular tablespace. As you add data, the data is written to one of the files which comprise the table's tablespace. (In Oracle 8, you can partition the table so parts of it reside in different tablespaces.)

Most of the hardware configurations for which we build applications have multiple disk drives. If the application has to access two pieces of data, retrieval will be quicker if the pieces of data reside on two different disk drives, since two different physical devices can do the read. Consequently, since an index and its table are always accessed in tandem, the table and index should be on different physical devices. Similarly, if tables are always accessed in the same query (for example, a customer and billing address), access will be faster if the tables reside on different physical devices.

So, one way to enhance performance is to assign the objects to tablespaces whose files reside on different disks.

It's intuitively obvious that tables and their indexes are always accessed together. Conversely, you can go crazy thinking about what tables will be used in conjunction with what other tables and figuring out a partitioning scheme. You need an "affinity analysis" to do this in any depth.

We actually started to write an affinity analysis using the Designer/2000 repository views. But we didn't think our development effort warranted such a detailed study. Instead, we used the application as the level of granularity for our analysis. We assigned the tables owned by each of five applications to a different tablespace, and the indexes to another tablespace. The naming convention used was:

- ?? a 3-letter designator for the application
- ?? followed by a dash

- ?? followed by TS for a table tablespace or IND for an index tablespace.

For example, the Supply application's tables were stored in the tablespace SUP\_TS, while the Procurement application's indexes were in PRO\_IND.

This gave us enough different tablespaces so that the DBA would have flexibility in deciding which file system each would be on, and associating files to the tablespaces. The tablespaces and their associated files don't actually have to be created until the build phase.

### Storagespaces

Storagespaces are Designer/2000 objects which group the parameters which can appear in a DDL STORAGE clause. They specify the physical storage requirements for each table or index created. We did not use the Database Design Wizard to associate storage specifications with tables. Getting this right would entail a more detailed and complete analysis of table and column usage than we felt 100% comfortable with at this time. Further, we didn't want to have to run the Wizard multiple times, just to assign different storage parameters to different sets of tables. Determining table-by-table storage requirements was part of the design and build phases.

### Analysis-to-Design Mapping

Analysis-level volumetrics are mapped into design-level properties by the Database Design Wizard.

Entity Property	Table Property
Initial Volume	Start Rows
Maximum Volume	End Rows

Attribute Property	Column Property
Percent Used - Initial	Initial Volume
Percent Used - Final	Final Volume

(The Annual Growth Rate and Average Volume are no longer used.)

### Designing Tables

The design phase is the time to further refine the data collected during analysis, and to specify additional properties to be used to create the physical database tables. The most universally used of these properties are found on the Table tab of the Edit Table dialog of the Data Diagrammer:

Number of Rows		Locking Sequence	
Start	End	<input type="checkbox"/>	<input checked="" type="checkbox"/> Create? <input type="checkbox"/> Journal?
<input type="text"/>	<input type="text"/>		
Storage Definition		Tablespace	
<input type="text"/>		<input type="text"/>	
Init Trans	Max Trans	Pct Free	Pct Used
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

### Percent Used and Percent Free

“Volatility” is the degree to which the contents of a particular row of a table change over the life of the row. Very low volatility rows, such as those in reference tables, are expected to be created with a set of values which rarely, if ever, change. Conversely, highly volatile rows are expected to be created only with a subset of the columns in the row filled in with their final values. More space will be required when a row that has already been stored in the database grows in length, either by updates of previously NULL fields, or by updates of existing values to longer values. Adding new rows or deleting rows from a table has no effect on row volatility.

The PCTFREE (percent free) and PCTUSED (percent used) parameters assigned to each table should directly reflect its volatility. PCTFREE “is used to set the percentage of a block to be reserved for possible updates to rows that already are contained in that block”. (Oracle7 Server Administrator’s Guide) PCTUSED, which DBA’s consider less critical, governs the amount of space which must be available in a block for a new row to be inserted there. These figures greatly influence the efficiency with which Oracle allocates space in the database to the tables.

In an attempt to get viable values for PCTFREE and PCTUSED, we followed a procedure like this:

1. The technical manager drew up a list of categories describing the different possible degrees of volatility of tables.
2. The database administrator specified the values of PCTFREE and PCTUSED that should be assigned to each category.
3. The application analysts determined which category each of their tables fit into, and plugged the associated PCTFREE and PCTUSED values into the table definition.

Our categories and values looked like this:

Pct Free	Pct Used	Usage
----------	----------	-------

Pct Free	Pct Used	Usage
2	95	Row contents rarely, if ever, change. Examples include load or unload tables, to which records are written once, and never updated; reference tables where field contents are very stable and/or consist almost entirely of numeric fields and/or consist almost entirely of NOT NULL fields.
10	88	Row contents may change, but values in most columns are supplied when the row is created. An example is a table loaded via SQL*Loader to which users may apply corrections only if a record fails to process correctly. Another example is a reference table in which the records are stable but some field contents, for example, a description, may change.
25	73	Row contents are changed when users enter values into columns that were left blank when the row was created. Once entered, values are probably stable.
40	57	Rows are subject to volatility of both previously NULL columns and value changes. Row contents change over the life span of the row. For example, additional comments may always be added.
60	38	A row is created with only a fraction of its values filled in. Over its life, more columns values are entered, and column contents are always subject to change. A table whose contents are subject to a lengthy approval cycle probably falls in this category.

Journal tables are not included in this list. By definition, rows in these tables never change. Consequently, Designer/2000 generates the DDL for journal tables with a PCTFREE of zero (0). If any other tables have rows which you can guarantee will never, ever be changed (even by clever developers using SQL\*Plus), feel free to use zero as the value for PCTFREE.

### Initial and Maximum Transaction Capacity

The transaction parameters govern the amount of space within a table’s header block that is reserved for simultaneous transactions to access rows in the same block of the table. The database initially allocates the space indicated by the Init Trans parameter. It then dynamically allocates enough space to accommodate up to Max Trans simultaneous transactions, as required. If any additional users try to access data in the same block,

they will have to wait until the previously started transactions terminate.

The default Max Trans value of 255 (for UNIX databases) is sufficient for most tables. However, the Init Trans value of 1 must be increased for parameter or similar tables, in which a relatively small number of rows is accessed by a relatively large number of users. The Init Trans value for these tables should be equal to the number of subsystem users who are likely to be accessing the table simultaneously.

### Start Rows and End Rows

The design phase is the time to refine the original estimates on table size, derived from the entity's initial and maximum volume. These properties become the start rows and end rows, respectively, of the corresponding table.

Our biggest problem was that we actually overestimated the number of rows in a table. This can easily occur when you convert legacy files containing non-normalized data into a normalized database with one-to-many relationships. For example, one legacy table allowed an inventory item to have up to four sources of supply. But how many rows actually had more than one? The customer had a rough idea ("not many"), but only examination of the legacy data yielded an answer usable for database sizing: 25% of the inventory items had two sources of supply, 10 items had three, only two items were actually available from four suppliers. This allowed us to accurately estimate the number of rows required in the child table.

If you are automating a previously manual process, the customer may have to examine pieces of paper to get prospective row counts. For example: how many items are typically on a purchase order? How many revisions are generally issued to a purchase request? Allow time for this information to be collected.

### Storage Definitions

A table grows dynamically. When you create it, it starts with an "initial extent", a fixed-size portion of the tablespace to which that table is assigned. As you add rows or update data, the table grows, to take up more room in the initially allocated extent. When that extent runs out of room, the table has to allocate its "next extent". All of the "next extents" of a table can either be the same size or each one can grow by "percentage increase" over the size of the previously allotted extent.

In Oracle 7, there is a hard and fast maximum number of extents that can comprise an index or table, based on the database block size. If an index or foreign key constraint runs out of extents, the DBA drops it and rebuilds it, a relatively minor annoyance. But if a table or the index of a primary key constraint runs out of extents, you're in worse trouble – the table data must be exported, the table dropped and rebuilt, including all the

constraints it involves.

Database design principles typically stress that all the rows in a table should fit in one extent. This optimizes the performance of full table scans, but has no effect on indexed access. We decided to design so that one year's worth of data (start rows) would fit into the one initial extent.

Clearly, it is in our best interests to get the extents right - or at least close. But, interestingly, extent size is not an integral part of table definition. It is stored in a "storage parameter" which can be associated with the table. This lets you define one storage parameter and, potentially, assign it to many tables with the same size and growth requirements.

Storage definitions are defined via their own property sheet in the Repository Object Navigator (RON):

<b>Storage Label</b>	
<b>Initial Extent</b>	
<b>Next Extent</b>	
<b>Min Extents</b>	
<b>Max Extents</b>	
<b>Percentage Increase</b>	

We tried to get some help in this area by running the [Database Table and Index Size Estimates](#) repository report. The most useful part of this report proved to be its explanation of the formulas it used, which are printed if you check *Include Help?*. We wanted to understand the formulas, so did a little more research.

**Initial Extent Size.** Extents are measured in the number of blocks they take up. The [Database Table and Index Size Estimates Report](#), the [Oracle Server Administrator's Guide](#), and most Oracle database administration books, include formulas to compute the amount of storage space which should be allocated for each table.

$$\text{bytes} = \text{block\_size} * (\#\_rows * \text{rows\_per\_block})$$

Looks easy, right? Hah!

To figure out the number of rows that will fit in a block, start by figuring out the row size, the amount of storage taken up by each row. This is equal to:

- ?? the size of the columns (in bytes) plus
- ?? the column overhead, which is 1 byte for every column shorter than 250 bytes, and 3 bytes for longer columns, plus
- ?? a row header of 3 bytes.

But what value do you use for column size – maximum length or average length? And do you counter in the percent used – either average or maximum? Our formulas were generous (and somewhat simplistic) and generally used the maximum length for columns, rather

than the average length, since data about average length was dicey, at best, and analysts were not religious about putting what they thought they knew into the repository. Further, we assumed that columns were always populated (100% usage).

Once you have the row size, you need to figure out how many rows will fit into a block. In principle, each block (on our UNIX DBMS) contains 2048 bytes. However, this doesn't take into account the block header size, as well as the overhead associated with each table's INITRANS and PCTFREE values.

- ?? The block size is 2048.
- ?? The block header size is 61 bytes (a constant) plus 23 bytes for each initial transaction (INITRANS).
- ?? For each percent of the block left free (PCTFREE), you also lose a certain amount of space in the block.

The information necessary to apply these formulas is stored in the Designer/2000 repository views ci\_columns and ci\_table\_definitions. Plugging these values into the formula for computing the amount of space that would be required for tables yields the SQL script in Figure 1.

The computed number of bytes must be rounded up to the next multiple of the block size, and the initial extent set to that size.

**Next Extent Size.** We were not at all scientific about computing the next extent size. The project DBA looked at a report listing the initial and "final" table size and told us what numbers we should use for our next extent. Next extent sizes ranged from just 10% of the initial extent size, for tables with low growth rates, up to 50% or more of the initial size.

**Minimum and Maximum Number of Extents.** The minimum and maximum number of extents specified for all our tables were the same – 1 as the minimum, and 121 as the maximum. 1 is recommended by most DBA handbooks; 121 is the maximum number of extents for database with 2K block sizes. In Oracle 8, this limit has become obsolete, and tables can be specified to AUTOEXTEND beyond the stated limit. However, production DBAs I've talked to are, so far, wary of letting tables AUTOEXTEND to their heart's content. Instead, they set the Max Extents to a few below the (previous) maximum, so they are warned before the problem gets out of hand. They also run regular reports on the status of their database, including how many extents each object is using. Some of those reports are given in the section of this document on building and testing your database.

**Percent Increase.** Most reference materials recommend using a percent increase (PCTINCREASE) of zero (0). This creates all "next extents" to be the same size, rather than growing each time, and makes your storage usage easier to track.

**How do you get there from here?** Now that we knew what numbers we needed, how did we set up our storage definitions? Storage definitions needed to be created for tables in the production database. Long before we had that database, we had to create tables in the development and test databases. We could not use the same storage parameters – there just wasn't enough room. For these databases, we used only five storage definitions for all the tables (and corresponding definitions for the indexes):

```
SELECT td.name, td.initial_number_of_rows init_rows,
       FLOOR(2048 - 61 - ((23 * NVL(td.initial_transaction,1)))
            - (2048 - 57 - 23 * NVL(td.initial_transaction,1)) *
td.percent_free/100)
       data_space,
       (SUM(DECODE(c.average_length, NULL, c.maximum_length, c.average_length))
        + 3 + COUNT(*) row_size,
       FLOOR((2048 - ((23 * NVL(td.initial_transaction,1)))
            - (2048 - 57 - 23 * NVL(td.initial_transaction,1)) *
td.percent_free/100)
        / (SUM(DECODE(c.average_length, NULL, c.maximum_length, c.average_length))
           + 3 + COUNT(*)) rows_per_block,
       CEIL (td.initial_number_of_rows /
            ((2048 - ((23 * NVL(td.initial_transaction,1)))
              - (2048 - 57 - 23 * NVL(td.initial_transaction,1)) *
td.percent_free/100)
            / (SUM(DECODE(c.average_length, NULL, c.maximum_length, c.average_length))
               + 3 + COUNT(*)))) number_of_blocks /* *2048 for number_of_bytes */
FROM ci_table_definitions td, ci_columns c
WHERE c.table_reference = td.id
      AND td.application_system_owned_by =98
      GROUP BY td.initial_transaction, td.id, td.percent_free, td.name,
              td.initial_number_of_rows
/
```

Figure 1. Computing table storage requirements

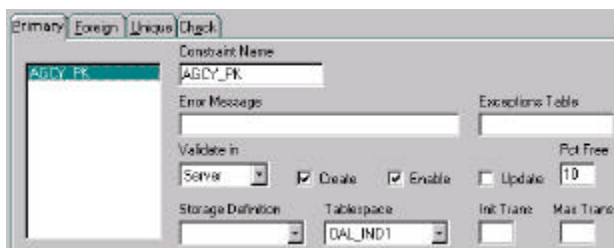
Storage Label	Init Extent
FLS_TEENSY_TABLE	10000
FLS_SMALL_TABLE	12000
FLS_MEDIUM_TABLE	14000
FLS_LARGE_TABLE	16000
FLS_HUGE_TABLE	18000

When the time came to create the production database, we needed a new set of storage definitions. We still tried to use the same storage definition for multiple tables, and gave the definitions labels like “FLS\_100K\_TABLE” or “FLS\_1MEG\_TABLE”. In retrospect, if I were doing this over, I would probably use the API to create one storage definition per table, based on the computed and standard parameter values, and to assign the definition to the corresponding table.

## Constraints

Constraints (which are created by Designer/2000 as “USING INDEX”) are subject to a subset of the parameters which apply to tables.

Take a look at Primary Key Constraints tab from the Constraints dialog of the Data Diagrammer:



For both primary and unique key constraints, properties which affect database structure include:

- ?? Percent Free (Pct Free)
- ?? Storage Definition
- ?? Tablespace
- ?? Initial Transactions (Init Trans)
- ?? Maximum Transactions (Max Trans).

### Percent Free

Percent free is the percentage of space that should be left free for potential updates of the values of columns which participate in constraints. In general:

- ?? Primary key constraints, by definition, are always mandatory and should not be updateable.
- ?? Unique key constraints may be mandatory or optional, updateable or non-updateable.

In principle, and according to Oracle Support, non-updateable primary key constraints can be assigned a PCTFREE of zero (0). However, our database administrator hates zeroes, so we used a PCTFREE of 1. This allows for the remote, but plausible possibility that programmers may, at some point, have to write code to

change a primary key value that, in the normal course of running the application, is not supposed to change.

By default, the PCTFREE parameter for creating indexes is ten (10) percent. We used that value for all other constraints.

### Storage Definition

Primary and unique key constraints should have associated storage parameters. Since the DDL for the constraints is created USING INDEX, the storage definition determines the initial and next extent sizes of the index.

If the storage definition is left blank, the constraint will be created with the same extent sizes as the table to which it applies. This will generally produce extents which are too large for the limited number of columns in the constraint. So, you need to develop formulas for estimating the extent sizes required, define appropriate storage parameters and plug these into the constraint definitions.

The formula for estimating the average index entry size is similar to the one used for calculating table row size:

- ?? As for tables, start with the sum of the column sizes.
- ?? The column overhead is 1 byte for every column shorter than 128 bytes, and 3 bytes for longer columns.
- ?? Add an entry header of 8 bytes.

The formula for calculating available space in the block is a little simpler, since it doesn’t need to take into account percent free.

You need an extra 5% of the entry size to account for additional branch blocks of the index, so:

$$\text{bytes} = \text{block\_size} * (\#\_rows * ((1.05 * (\text{row\_size} + 8 + \#\_fields))) / \#\_entries\_per\_block$$

The information necessary to apply this formula is stored in the Designer/2000 repository views:

- ?? ci\_table\_definitions for tables
- ?? ci\_key\_components for constraints
- ?? ci\_columns for columns.

The formula developed is given in Figure 2.

## Indexes and Foreign Keys

Take a look at the Index tab of the Edit Table dialog of the Data Diagrammer.

It allows you to specify the same parameters as are found on the Primary and Unique Key Constraints tabs:

- ?? Percent Free (Pct Free)
- ?? Storage Definition

```

SET HEADSEP \
COLUMN initial_number_of_rows FORMAT 9,999,999 HEADING 'START ROWS'
COLUMN max_len FORMAT 9,999 HEADING 'MAX\LEN'
COLUMN init_extent FORMAT 999,999,999 HEADING 'INIT\EXTENT'
COLUMN total_extents FORMAT 999,999,999 HEADING 'TOTAL\EXTENTS'
SELECT
  SUBSTR(td.name,1,30) table_name, SUBSTR(kc.name,1,30) constraint_name,
  td.initial_number_of_rows,
  SUM(c.maximum_length) max_len,
  2048 *
  (1.05 * (td.initial_number_of_rows * (SUM(c.maximum_length) + 8 + COUNT(*))) /
    FLOOR ((1700 / (SUM(c.maximum_length) + 8 + COUNT(*))) *
      (SUM(c.maximum_length) + 8 + COUNT(*))))
  init_extent
FROM ci_table_definitions td, ci_columns c, ci_key_constraints kc
WHERE c.table_reference = td.id
  AND td.element_type_name = 'TAB'
  AND kc.table_reference = td.id
  AND td.application_system_owned_by = 98
GROUP BY td.name, td.initial_number_of_rows, kc.name
/

```

**Figure 2. Computing constraint storage requirements**

- ?? Tablespace
- ?? Initial Transactions (Init Trans)
- ?? Maximum Transactions (Max Trans).

<b>Parallel?</b>	
<b>Degree</b>	

Insofar as database structure is concerned, indexes are subject to the rules very similar to those that apply to constraints. The formula for computing constraint space usage can be reused, simply by substituting the repository view `ci_index_entries` for `ci_key_constraints`.

While some members of the team are figuring out database structure, others are developing reports and queries, and attempting to tune their performance. Such tuning may include creating additional indexes, not corresponding to foreign key constraints. Be sure your procedures mandate running the index storage usage script before the index can be created. You will probably need another version of this script, which allows the analysts to specify the name of a single table or index for which storage requirements need to be calculated.

The Index tab also includes entries for specifying the degree of parallelism of the index. Index parallelism is used only for creating indexes via the `CREATE INDEX` command, and is not relevant to accessing the table via the index. Indexes are typically dropped before a large scale batch load, and recreated when the load is complete, to enhance performance. Since our application did not have involve such loads, we did not use these entries.

## Parallel Queries

If your database server has multiple processors, Oracle can be configured to use its parallel query option. A parallel query means that the DBMS can start multiple “query servers” to allow `SELECT` commands to simultaneously retrieve multiple rows. Parallel query servers can process either simple queries or sub-queries embedded in `UPDATE` or `DELETE` commands.

The Storage section of the Table property sheet in the RON includes properties pertaining to parallel queries:

In a multi-CPU machine, the `init.ora` parameter `optimizer_percent_parallel` probably come configured with the default degree of parallelism. (I think it was 5 for our HP-T500). That means that unless a query which would use the parallel option overrides the default, the default is used. In Designer/2000, you can override the default degree of parallelism for a particular table by changing `Parallel?` to True and entering a different degree of parallelism. To override both the database default and the table default, you can use a hint on a particular query.

We investigated tuning the degree of parallelism as part of physical design. Unfortunately, such tuning depends greatly on system usage patterns and total system load, so we decided not to apply any of the parallel options (except to leave the database default) until after the system had been in operation and we knew what the usage patterns really were.

Therefore, we just left the query specifications for most tables at the default – use the default degree of parallelism. The production DBA knows that she should be on the lookout for increasing or decreasing the degree of parallelism allowed for particular tables or queries.

The one thing we could do during the design phase was determine which tables should not use parallel queries at all. The criteria are similar to those you apply for determining when to add an index to a table – small tables don’t need it. So, the `Parallel?` property was changed to False, to generate a `NOPARALLEL` clause in the DDL of all tables with fewer than 1000 rows to start.

## Build and Test

Okay, you gave it your best shot. You got detailed sizing estimates from users, developers analyzed table

usage for their system, all the values were inserted into the repository and DDL generated. The production database instance was created, the DDL run, and the tables populated with a combination of new reference data, converted legacy data, and test data.

What can you do to check how well you did?

Unfortunately, not much. The usage patterns have not yet been established, unless you get the users

### Scripts to Check Extent Usage

Two aspects of extent usage must be checked:

- ?? Does the object as initially populated have too many extents? (Our goal was to start with one extent per table or index.)
- ?? Does the table take up too much space; that is, is a substantial part of the initial extent allocated not being used? (Yes, hardware is cheap, but not that cheap.)

The project DBA wrote SQL scripts, interrogating the system catalog, to determine whether the table or index was taking up more than one extent. For indexes, the script read:

```
COL used FORMAT 9,999,999 HEADING 'Used
Space'
COL extent_id FORMAT 999 HEADING '#\Ext'
COL initial_extent FORMAT 9,999,999
HEADING 'Initial\Extent'
SELECT ai.index_name, ai.initial_extent,
       ai.next_extent, de.extent_id,
       ai.initial_extent +
       (next_extent * de.extent_id) used
FROM all_indexes ai, dba_extents de
WHERE ai.index_name = de.segment_name
AND ai.owner = '&OWNER'
AND de.extent_id > 0
AND de.extent_id =
      (SELECT MAX(de2.extent_id)
       FROM dba_extents de2
       WHERE de2.segment_name =
             de.segment_name)
```

The extent identifier (extent\_id) starts with zero (0), so any extent\_id of 1 or higher means that the table has extended. By multiplying the number of “extra” extents times the size of the next extent, we could see how much we needed to increase the initial extent by.

The converse was seeing which tables were allocated with too large an initial extent. For that we used a script like:

```
TITLE 'TABLES THAT HAVE NOT EXTENDED'
COL blocks FORMAT 9,999 HEADING 'Blk'
COL empty_blocks format 99,999 HEADING
'EMP BLK'
COL initial_extent FORMAT 999,999,999
HEADING 'INI EXTENT'

SELECT at.table_name, at.initial_extent,
       at.num_rows * at.avg_row_len
Used,
       (at.blocks * at.avg_space) +
       (at.empty_blocks * 2048) Free,
       at.blocks, at.empty_blocks
```

```
FROM all_tables at, dba_extents de
WHERE at.table_name = de.segment_name
AND at.owner = '&OWNER'
AND at.num_rows > 0
AND 0 = (SELECT MAX(de2.extent_id)
        FROM dba_extents de2
        WHERE de2.segment_name =
              de.segment_name)
```

This script only worked for tables. The project DBA never did figure out the corresponding script for indexes and constraints. If anyone has an appropriate script, please let me know.

## Conclusion

Physical database design usually means laying out your tables, columns, database-specific data types, and other implementation-specific artifacts. If it's going to be done correctly, it needs to go one layer deeper, to how these database components are actually stored. I think this paper goes a nice distance toward accomplishing that goal. What we need to do is get some of the techniques discussed here, both technical and staff, incorporated into our development methodologies.